

AD-A192 848 IMPROVING THE PERFORMANCE OF AI ALGORITHMS(U) AUBURN
UNIV AL DEPT OF COMPUTER SCIENCE AND ENGINEERING

1/2

UNIV AL DEPT OF COMPUTER SCIENCE AND ENGINEERING
C M PANCAKE SEP 87 SCCE-PDP/85-48 RADC-TR-87-131

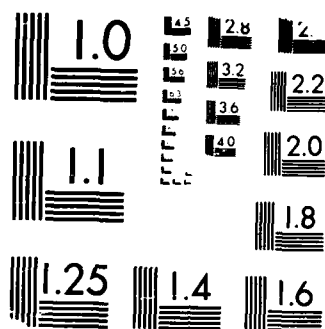
UNCLASSIFIED F30602-81-C-0193

F/G 12/9

NL

五

三



MICROCOPY RESOLUTION TEST CHART
 10101 1-61965

AD-A192 848

DTIC FILE COPY

(4)

RADC-TR-87-131
Final Technical Report
September 1987



IMPROVING THE PERFORMANCE OF AI ALGORITHMS

Auburn University

**Sponsored by
Strategic Defense Initiative Office**

**DTIC
ELECTE
MAR 25 1988**
S D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.

**ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700**

88 3 25 02 8

IMPROVING THE PERFORMANCE OF AI ALGORITHMS

Cherri M. Pancake

Contractor: Auburn University
Contract Number: F30602-81-C-0193
Effective Date of Contract: 15 May 1985
Contract Expiration Date: 30 April 1987
Short Title of Work: Complex of AI Algorithms Program
Period of Work Covered: May 85 - Apr 87

Principal Investigator: Dr. Charles R. Vick
Phone: (205) 826-4330

Project Engineer: Mr. Donald J. Gondek
Phone: (315) 330-4833

Approved for public release; distribution unlimited.

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Donald Gondek (RADC/COES) Griffiss AFB NY 13441-5700 under Contract F30602-81-C-0193.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SCEE-PDP/85-48			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-131		
6a. NAME OF PERFORMING ORGANIZATION Auburn University		6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)		
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science & Engineering 107 Dunstan Hall Auburn University AL 36849-3501			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Strategic Defense Initiative Office (SEIO)		8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0193		
8c. ADDRESS (City, State, and ZIP Code) Office of the Secretary of Defense Wash DC 20301-7100			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 65223C	PROJECT NO. B413	TASK NO. 05
			WORK UNIT ACCESSION NO. P2		
11. TITLE (Include Security Classification) IMPROVING THE PERFORMANCE OF AI ALGORITHMS					
12. PERSONAL AUTHOR(S) Cherri M. Pancake					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM May 85 TO Apr 87		14. DATE OF REPORT (Year, Month, Day) September 1987	
				15. PAGE COUNT 120	
16. SUPPLEMENTARY NOTATION Graduate research assistants who contributed to the preparation of this report: Laura F. Henry, Paula Sue Utter, Gregory K. Whitefield					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD GROUP SUB-GROUP			Artificial Intelligence (AI) Algorithms, Improving Software		
12 05			Performance, Program Behavior, Predicting Performance,		
12 07			Optimization, Programming Languages, AI Software. (cont'd)		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) (ARTIFICIAL INTELLIGENCE) → The feasibility of improving the efficiency of AI software using available systems and methodologies is addressed. By modeling program behavior as a series of concurrent problem solution systems, it is possible to isolate the inefficiencies inherent in the implementation scheme for those due to conceptual difficulties or inadequacies in the underlying physical system. → The processing environment selected for the implementation of AI software effectively establishes a computational paradigm which shapes the development and ultimate performance of any program executing within it. Sequential environments view the underlying architecture as von Neuman and approach a problem in terms of the Turing Model of Computation, while applicative environments exemplify the recursion theory approach. Established optimization techniques are intimately tied to the computational model and cannot be transported from one environment to the other with ease or efficiency. (Cont'd)					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL DONALD J. GONDEK			22b. TELEPHONE (Include Area Code) (315) 330-4833		22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

UNCLASSIFIED

18. (Cont'd). Programming Language Techniques

19. (Cont'd). Since some AI tasks are inherently sequential and others inherently recursive, no single processing system can facilitate uniformly optimum performance. The concept of "environment spanning" is suggested as a means of maximizing program optimizability by allowing the assignment of subproblems individually to whatever processing system offers the best chance for automatic improvement. Three mechanisms for implementing spanned environments are presented: parallel environments, multitasked environments, and intersequenced sub-environment modules.



Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

UNCLASSIFIED

Contents

List of Figures	iv
1 Introduction	1
2 Implementation of AI Algorithms	6
2.1 Fundamentals of Program Behavior	8
2.2 Predicting Performance	11
2.3 Improving Performance	16
3 Limitations on Potential for Optimization	23
3.1 Structure of the Implementation Solution System	27
3.2 Characteristics of Sequential Processing	33
3.3 Optimization in the Sequential Environment	37
3.4 Characteristics of Applicative Processing	41
3.5 Optimization in the Applicative Environment	47
4 A Strategy for Maximizing Optimization Potential	51
4.1 Evaluating Processing Environments	52
4.2 The Environment Spanning Strategy	57
4.3 Environment Spanning Implementations	62
5 Conclusions	69
References	72
Appendix A. Optimization Techniques for Sequential Environments	74
A.1 Typical Optimization Techniques	76
A.2 Potential for Optimization	82
Appendix B. Optimization Techniques for Applicative Environments	90
B.1 Typical Optimization Techniques	90
B.2 Potential for Optimization	94

List of Figures

Figure 1	Program Performance Modeled as Concurrent Problem Solution Systems	2
Figure 2	Predicting Program Efficacy	9
Figure 3	Predicting Program Efficiency	14
Figure 4	Five Common Optimizing Transformations	20
Figure 5	Implementation Stages at Which Optimizing Transformations May Be Applied	21
Figure 6	Abstract Machines in the Implementation Solution System	29
Figure 7	Programming Language "Hierarchy"	31
Figure 8	System Support Layering in Sequential Processing Environments	35
Figure 9	System Support Layering in Applicative Environments Hosted on Sequential Processors	44
Figure 10	System Support Layering in Applicative Environments Hosted on Non-Sequential Processors	45
Figure 11	Gabriel's Tak Benchmark Comparing LISP with Procedural Languages	55
Figure 12	Environment Spanning Using Parallel Processors	64
Figure 13	Environment Spanning Using Multitasked Processing	65
Figure 14	Environment Spanning Using Intersequenced Modules	67
Figure 15	Examples of Expression Simplification Techniques	78

Figure 16	Examples of Code Rearrangement Techniques	80
Figure 17	Estimated Effects of Optimization	86
Figure 18	Arklam's Benchmarks on the Effects of Optimization	87
Figure 19	Wulf's Quantification of the Effects of Optimizing Techniques	88
Figure 20	Gabriel's Benchmarks on the Effects of Optimization	95

1 Introduction

The complex interrelationships of computer systems within the BM/C³ setting impose stringent requirements on their performance. They must reliably produce correct results within a minimal period of time and without exorbitant demands upon external resources. At the same time, they must be capable of flexible and dynamic response to changes in the processing environment, adapting quickly to fluctuations in communications, threat assessment, resource availability, and so forth. This need for intelligent and adaptable behavior indicates that the integration of artificial intelligence algorithms may provide significant enhancements in the behavior of BM/C³ systems.

The history of poor performance demonstrated by past AI systems has made real-time behavior an issue of concern. Can optimization techniques be systematically applied to AI programs in order to bring their performance up to real-time standards? Does any such improvement presuppose the development of new hardware and/or software capabilities? The present report discusses the feasibility of optimizing AI algorithms using currently available resources and methodologies and proposes a strategy for maximizing improvement potential.

The issues involved in optimizing AI programs can best be understood if we model the performance of an AI algorithm in a real-time situation as a series of problem solution systems operating concurrently at different levels of abstraction (see Figure 1). At

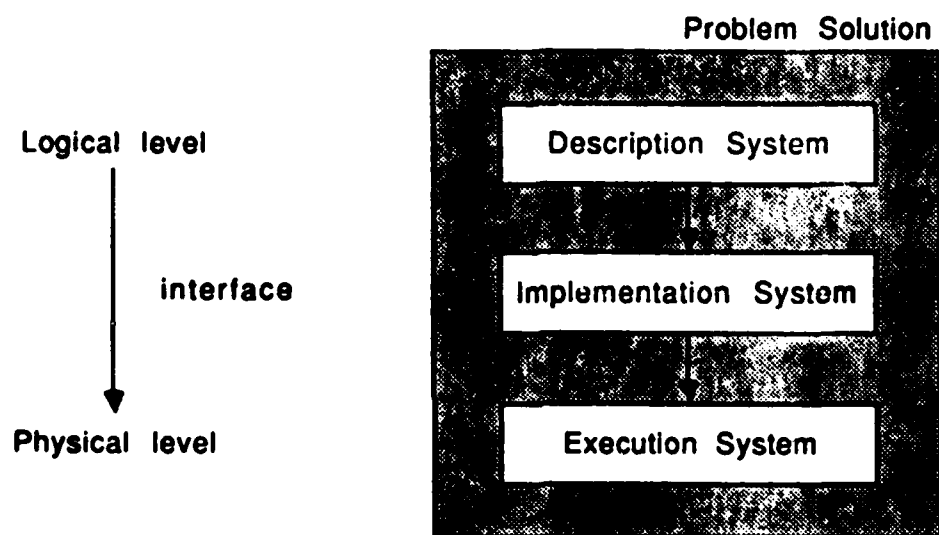


Figure 1. Program Performance Modeled as Concurrent Problem Solution Systems

the highest level is a logical description of the problem and the steps required to solve it. At the lowest level is a physical system, the architecture configuration executing the solution. Between the two lies an implementation system which provides the interface between logical conceptualization and physical reality.

An AI algorithm can be optimized at any or all of these levels. The logical system is improved by devising a more efficient conceptual solution to the problem. This may involve a decrease in the amount of processing required, as is the case with the development of search tree pruning techniques, more effective heuristic functions for evaluating progress toward a goal, or methods of minimizing the need for data retrieval. The logical solution may also be streamlined by the development of faster processing techniques such as improved methods of discrimination or better data structures for problem representation.

The execution solution system is improved by enhancements to the physical resources. The architectural configuration processes each operation by first analyzing the memory, processor, and interconnection network elements required and then controlling the sequencing of those elements. Improvements include the use of faster processors, lookahead control capabilities, and configurations allowing speedier data retrieval, as well as the most obvious enhancement, the distribution of execution over a series of parallel processors.

The intervening system, the implementation, serves to bridge the

"semantic gap" between the expression of the problem as a set of conceptual relationships and as a series of operations to compute those relations. A certain degree of inefficiency is, of course, inherent in any situation where a logical solution must be mapped to a physical one. By viewing the interface system as a separate entity, however, we can attempt to improve its transformations as a means of generally enhancing the performance of the algorithm. This process involves isolating those aspects of the description system which have most impact on execution and establishing a means of minimizing transformational inefficiencies.

Most recent research efforts in the improvement of AI programs have been devoted to the description and execution solution systems. The current study addresses the feasibility of optimization at the implementation system level by exploring the logical/physical interface and its performance implications. Chapter 2 presents an overview of the implementation solution system and describes the issues involved in assessing and enhancing program performance. This establishes a foundation for the next chapter, which discusses the limitations imposed by the processing environment and presents comparative studies of program optimization in two general environments, sequential and applicative. Chapter 4 introduces the concept of environment spanning, a strategy which seeks to maximize the "optimizability" of AI algorithms by partitioning programs into segments for coordinated processing in a heterogeneous environment. A final chapter summarizes the conclusions of the study. The appendices

provide supplementary information on optimization in the sequential and applicative environments by cataloging the optimizing transformations typically applied in each and summarizing empirical studies of the effects of optimization on program performance.

2 Implementation of AI Algorithms

The artificial intelligence algorithms developed for battle management applications are subject to strict prerequisites. They involve not only an unusual degree of numerical computation, but also rigid performance constraints imposed by the real-time nature of BM/C³ systems. AI programs in this setting are sophisticated and often large in scale, requiring extensive supplementary databases which guide inference and discrimination systems or assist in calculating heuristic evaluations of goal proximity, resource adequacy, trajectory fit, etc. The size and complexity of these programs relegates them to a long-term development framework. This has the effect of imposing the additional restriction that the nature and amount of processing required be predictable, at least to the extent that there is some assurance the algorithm can eventually perform in real-time situations. It must be possible to simulate or otherwise analyze the selected implementation strategy, predict the impact of external system conditions, and guarantee that performance can meet stringent time constraints.

As outlined in the preceding chapter, the implementation system provides an interface between the conceptualization of the problem solution and the hardware-specific processing. It encompasses a number of levels and types of elements, including the goals and subgoals selected for the implementation, the programming language in

which the solution is expressed, the translating systems used to map the program to target machine instructions, and any run-time environment layers which isolate the user program from the hardware. All of these influence run-time behavior, but the nature and extent of their effects vary significantly.

The feasibility of performance improvement in the implementation system depends to a great extent on two factors: (1) the ability to accurately predict system performance, particularly in the sense of identifying those features responsible for degradation; and (2) the capability of applying some sort of optimizing transformations which enhance the predicted behavior. This chapter addresses these issues by examining the general nature of program behavior, behavior prediction, and optimization techniques. System characteristics which impose limitations on the extent of program improvement feasible within an AI framework are then presented.

Certain assumptions have been made as to terminology. The term translator refers globally to system software which maps a program in one language (the source) to another (the target). These include -- but are not limited to -- compilers, interpreters, macroprocessors, and assemblers. Similarly, a translation is any mapping from a source to a target language, typically a one-to-many transformation (i.e., from a "higher" to a "lower" level). Implementation is used in reference to any activity within the implementation solution system; where no ambiguity results, the terms program and implementation system appear interchangeably. Finally, behavior and performance are

used synonymously to describe the observable aspects of execution such as speed and resource utilization (as opposed to non-observable features such as fault tolerance and correctness).

2.1 Fundamentals of Program Behavior

Although the notion of program behavior can obviously be approached from a number of viewpoints, it is convenient for our purposes to group the elements of the implementation solution system into two classes. The first includes those factors relating to the processes by which the system is established, while the second encompasses those influencing the way in which the mature system behaves. A corresponding distinction is drawn between efficacy and efficiency as performance metrics.

Implementation efficacy measures the "effectiveness" of the system: an effectual program correctly produces the desired effect (without regard to exactly how the effect is achieved). Efficacy is determined from the processes used to create the implementation rather than from consideration of the way in which it functions.

The evolution of an implementation system is typically viewed as a sequence of phases (see Figure 2): the definition of requirements, establishing of specifications, design and development of the source program, and translation into target form. Each transition from one phase to the next requires an expansion of the problem representation from a relatively abstract to a more concrete level. Associated with

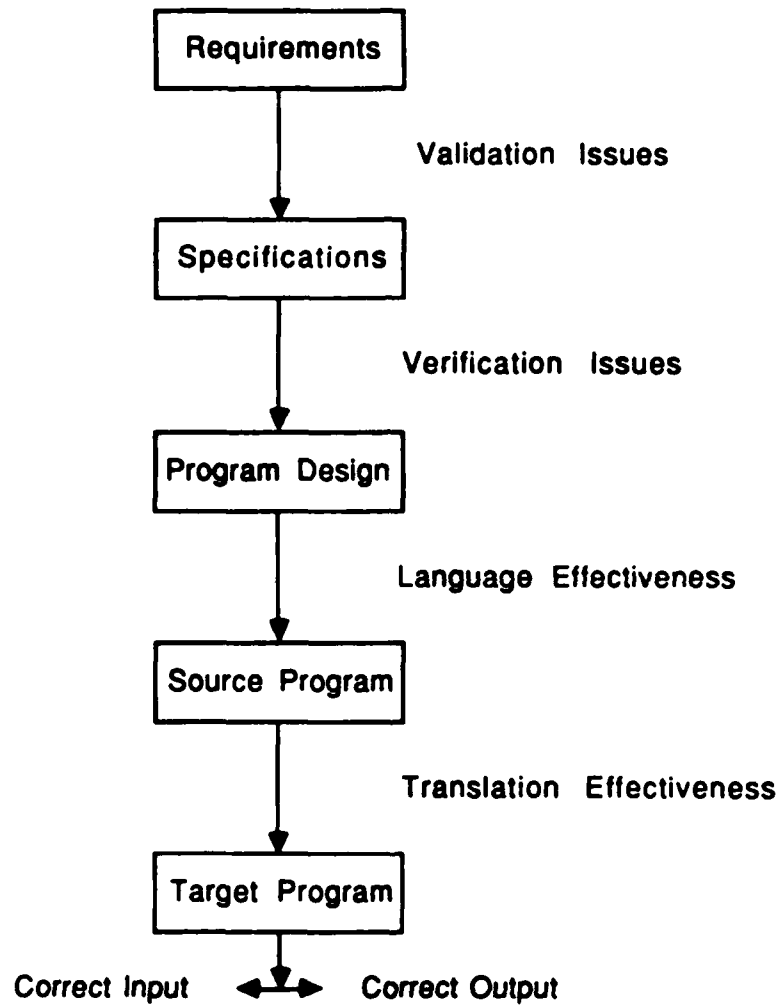


Figure 2. Predicting Program Efficacy

each expansion is a series of constraints intended to guarantee the fidelity and reliability of the mapping transformation. The constraints imposed at early phases of development are generally viewed as issues of validation (does the implementation do what is intended?) and verification (does it produce the effect correctly?). Efficacy factors such as these are of major concern in software engineering, but are beyond the scope of the present work.

The selection of a source language and the translation of the resulting program are relevant to any discussion of performance improvement, but in terms of efficacy their influence is difficult to measure. For reasons which should be obvious, subsequent sections assume that: (1) the implementation languages selected provide for a suitable representation of the algorithm; (2) the program faithfully describes the problem solution; and (3) the translators available provide adequate and reliable mappings from source to object code.

In contrast to efficacy, implementation efficiency measures solution competency: an efficient program executes the task with a minimum expenditure of physical resources (without concern for whether or not the task correctly solves the problem). Efficiency issues, therefore, derive from the manner in which the implementation functions during program execution.

The behavior of an implementation system can be characterized as a hierarchy of abstract or virtual machines representing logical levels of functionality (described in detail in later sections). The interface between one machine and the next consists of a one-to-many

mapping between instruction sets, with the final or lowest interface providing a mapping to the machine code of the target architecture. Implementation efficiency is a composite measure of the efficiency of all these interfaces. It reflects the appropriateness of the target architecture to the task, as well as the adequacy of the transformations.

2.2 Predicting Performance

The developer of any real-time computational system makes an implicit assumption that behavior can be predicted prior to program execution and that efficiency is not only an attainable, but also a measurable goal. It should be noted that execution-time behavior in itself may not be a sufficient criterion for assessing program performance. If the program forms part of a complex software system, additional factors such as reliability, maintainability, modifiability, and transportability must be taken into account; in some cases these requirements are in direct conflict with the goals of efficacy and/or efficiency.

A program's behavior can be characterized on a variety of levels, including execution speed, amount of memory occupied by program instructions, data storage requirements, number and type of operations performed, need for exclusive access to shared devices, and

so forth.¹ Such measures are difficult to quantify and tend to be distressingly sensitive to specific run-time conditions. In general, however, they may be categorized as influencing three facets of program efficiency: conciseness, speed, and throughput.

Conciseness measures the main and auxiliary memory space required by the program for both code storage and data representation. A program maximizes conciseness by occupying a minimal amount of space. Speed provides a similar metric, expressed in terms of the amount of time required for processing. Since this is influenced by how much of the program is resident in main memory versus how much must be swapped in and out of auxiliary storage, conciseness and speed are inextricably related. The third facet, throughput, assesses program productivity or output vis-a-vis input. This measure is inversely related to the number of interrupts causing the suspension of program execution. Although somewhat related to speed (a slow program is more likely to be interrupted than a fast one), throughput is more representative of external factors, such as the program's requirements for system resources, file structure and data redundancy, or available error handling facilities.

Although the nature of conciseness, speed, and throughput are easily understood, their evaluation is difficult. The crux of the

¹ The distinctions are not always clear; some program environments, for example, do not differentiate between program data (instructions) and problem data (variable storage). For purposes of clarity, these issues are deferred until later chapters.

problem is the identification of suitable standards for testing efficiency. A software developer needs to predict the behavior of a proposed implementation assuming it is designed and carried out with "average" programming skill and is executed utilizing "average" data on a system running under an "average" load. The use of the criterion "average", however, while perfectly natural to the human designer, is often a statistical impossibility in terms of the computational system. The assessment of program performance is commonly reduced to a discrimination process which selectively isolates specific examples (benchmarks) from the program solution space, further restricts those examples by the application of selected input (test cases), and executes them under selected system conditions, often simulated (see Figure 3). The resulting measurements are at best a rough approximation and at worst a gross misrepresentation of real-time conditions.

The selection of test programs commonly entails the use of an evaluation suite. This may be composed of task-specific, application-specific, or naturally-occurring benchmarks, or any combination of the three. Task-specific benchmarks, which isolate particular and presumably typical program activities for individual measurement, are the performance evaluator's version of unit-level testing. Although they can be a valuable source of data on speed and throughput, task-specific tests are often misleading because they cannot reflect system interactions or load conditions. Furthermore, they are inordinately susceptible to "edge effects", or the non-representative results which occur when a program slightly exceeds or barely misses

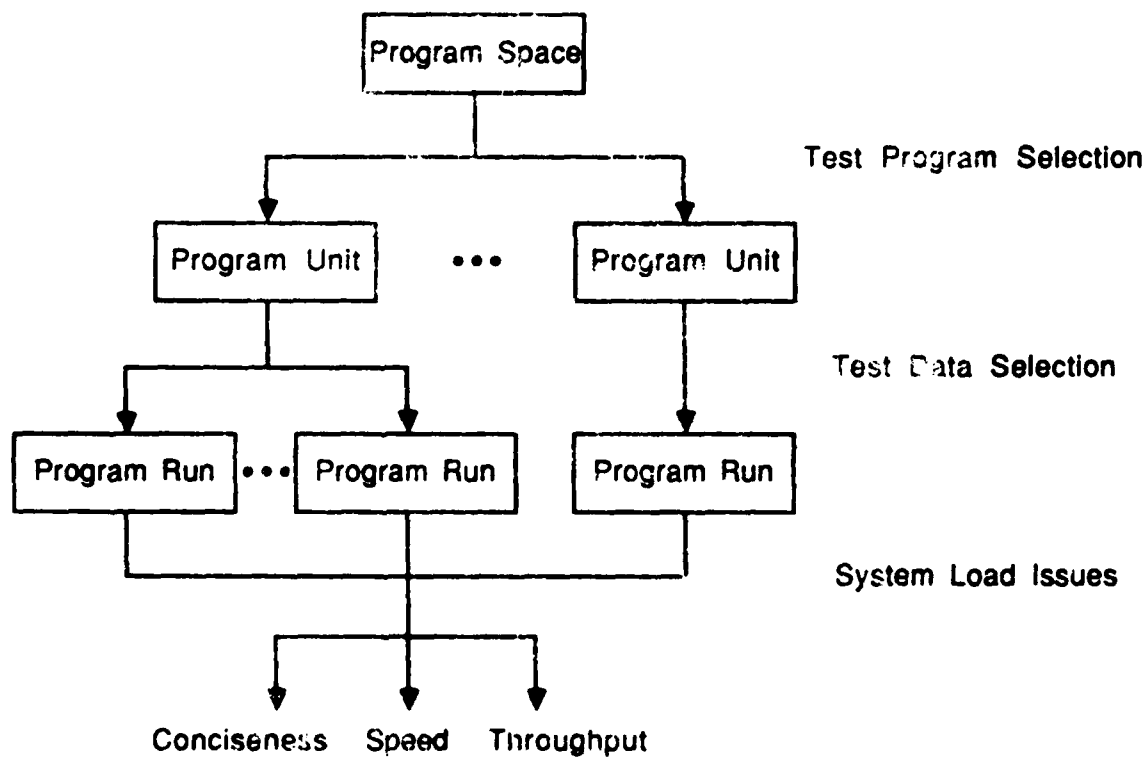


Figure 3. Predicting Program Efficiency

critical limits (such as memory page size or I/O transmission bounds).

To a certain extent, application-specific benchmarks, which attempt to provide more realistic measurements by modeling the proposed solution on some reduced scale, compensate for these drawbacks. They are prone to the biases typical of any modeling situation, however, notably oversimplification of the problem and the failure to accurately portray the effects of system load. Since they execute on a reduced scale, there is also a tendency to atypical behavior, such as a distorted view of initiation versus execution costs.

Naturally-occurring code may also be selected for benchmark analysis. In this case, existing programs which solve related problems are utilized in an attempt to approximate "real-life" processing conditions. Although these are somewhat better than other benchmarks in reflecting the effects of system load, they are often biased toward a specific problem or implementation system. They are also more likely to reflect individual levels of programming skill and/or system-dependent optimizations.

Occasionally, criteria other than benchmark suites are used to assess program behavior. The most common approach is to evaluate the mappings which provide the interface from one abstract machine to the next by identifying the number and type of instructions generated at each level and ultimately quantifying the time required to execute the physical machine instructions. The primary drawback of this methodology is that it cannot adequately reflect external run-time conditions influencing program behavior, such as system load, resource contention

problems, and communication blockages. Like task-specific benchmark tests, this type of measurement is also susceptible to edge effects.

It is obvious that no single methodology for evaluating program behavior accurately portrays the interrelationships among conciseness, speed, and throughput. The assessment of program behavior is still a black art, and few guidelines are generally applicable. These caveats should be borne in mind when later sections discuss the results of benchmark analysis.

2.3 Improving Performance

The performance requirements inherent in real-time computation systems have engendered a widespread interest in the development of techniques for program optimization. In general, optimization refers to the transformation of a program implementation in order to improve its execution-time performance. The term is often applied loosely to any translator which makes a significant effort to generate efficient target code; it is, however, better applied to specific attempts to rearrange or alter program operations so that the target program is more efficient than that generated by a direct translation.

The term optimization is misleading for several reasons. First, the notion of optimality is imprecise. No known metric suffices to describe behavior dynamics and, as indicated in previous sections, performance in itself may not be a suitable criterion for evaluating program "goodness". Furthermore, the term optimization implies that a

unique optimal solution exists and that it can be recognized as such. This is extremely unlikely, in view of the complex interrelationships among individual performance characteristics; optimizing transformations can rarely be applied without ambiguity, and the improvement of one aspect of a program's behavior can have adverse effects on others. Finally, most existing techniques are applied on the basis of pre-execution program analysis, restricting their activities to those portions of the program which are not overly dependent on run-time values.

In spite of its inappropriateness, however, optimization is the term most generally used in reference to performance improvement. The present report will use the words optimization, improvement, and enhancement interchangeably, with the understanding that they represent a relative rather than an absolute goal.

Central to the idea of optimization is the concept of program equivalence. Since any number of programs can be devised to produce identical run-time results, the goal of program improvement is the generation of the most efficient means of achieving the desired results. Optimizing transformations thus represent automated attempts to improve upon the programmer's description of the algorithm. Such alterations are important in order to compensate for the inefficiencies inherent in the use of high-level languages, which often suppress those details of the object language having most influence on program performance.

The implementation of program improvements presupposes the formulation of a set of transformations which will produce a program equivalent to the original. Each such transformation is described in terms of the relationships between program elements which are necessary preconditions, in combination with a meaning-preserving transformation rule. Any constraints governing the order of applying transformations must also be supplied. The situation is complicated by the fact that relatively few optimizations are finite. If a transformation can be repeatedly applied without ever reaching a point where it is no longer possible to apply it, artificial boundary conditions must be established to terminate the process. Additional safeguards may need to be instituted in order to prevent conflicts between individual transformation rules.

The concept of meaning-preserving transformations can most easily be understood by viewing the execution of a particular program as a sequence of actions ($A_1 \dots A_n$). For reasons which will be made clear in later chapters, these actions should be considered abstractly and not equated with program instructions; since each program action is explicitly represented, there are no control constructs (e.g., branches, loops, etc.) in this representation. Figure 4 illustrates the application of five commonly applied optimizing transformations to such an execution sequence. Note that some of the improvements may be contradictory. For example, the replacement of an action by a faster equivalent may increase program storage space as it shortens execution time. Similarly, the elimination of redundant calculations saves time

and program storage but may increase requirements for temporary data storage.

Improvement strategies can be classified according to a variety of criteria. As indicated in Figure 4, it is common to group transformations according to the type of improvement effects (e.g., reduction in execution time, reduction of instruction storage, reduction of data storage, reduction of I/O interfaces, etc.). Other categorizations are based on technique applicability (machine dependent or independent optimization), scope of improvement efforts (local, global, interprocedural, etc.), number of applications (finite vs. infinite transformations), and of course the particular type of technique used (e.g., expression simplification, code rearrangement, operation frequency reduction, peephole optimization).

An alternative approach is to view optimizations in terms of the implementation stages at which they are performed (Figure 5). Source-level transformations are applied by a preprocessor which generates an altered version of the source program, allowing machine independent (but language dependent) improvement strategies. Object-level transformations, applied by a postprocessor to the machine code generated by the translator, are conversely machine dependent and language independent. Most current implementations of optimizers, however, are incorporated in compiler systems and operate on some intermediate form of program representation. The transformations are of varying degrees of language/machine dependence, according to whether they are applied during the syntactic analysis, semantic


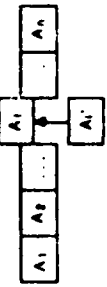


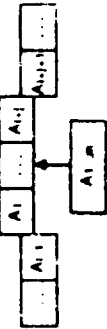





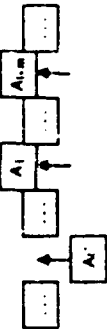

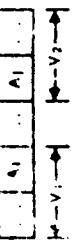

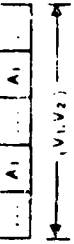
Indicative Condition	Transformation Rule	Result
 <p>A_1 is equivalent to A_2 but faster</p>	 <p>Replace original action by faster action</p>	 <p>Execution time is shortened</p>
 <p>Consequence of n actions is equivalent to sequence of m actions, where $m < n$</p>	 <p>Replace original sequence by shorter sequence</p>	 <p>Program storage space is reduced</p>
 <p>A_1 exactly duplicates A_2 (with no side effects)</p>	 <p>Eliminate redundant action</p>	 <p>Execution time is shortened and program space is reduced</p>
 <p>A_1 appears in i repetitions, but is invariant</p>	 <p>Replace all original occurrences by one placed outside the repetitions</p>	 <p>Execution time is shortened and program space is reduced</p>
 <p>Data element V_1 not needed after A_1; V_2 not needed before A_1</p>	 <p>Overlay V_1 and V_2 in storage</p>	 <p>Data storage space is reduced</p>

Figure 4. Five Common Optimizing Transformations

Pre-Translation
Stage

SOURCE PROGRAM

Preprocessor

Lexical and Syntactic Analyzers

PARSE TREE

Semantic Analyzer

Translation
Stage

INTERMEDIATE CODE

Code Generator

OBJECT CODE

Post-Translation
Stage

Postprocessor

EXECUTABLE CODE

Figure 5. Implementation Stages at Which
Optimizing Transformations May Be Applied

attribution, target code generation, or peephole optimization (last stage of code generation) phase.

To be effective, an optimization technique must guarantee improved performance in all possible execution cases and must not change either program behavior or results, even if execution is abnormally terminated. In addition, it should be cost effective in terms of the time required to perform the transformation compared to the execution-time improvement which will be realized. Although ideally it should be possible to apply transformations consecutively with no ambiguity or conflict, in practice most improvement techniques are complex, of uncertain duration, of limited applicability, and often contradictory in nature and results.

3 Limitations on the Potential for Optimization

Although the nature of optimizing transformations poses some inherent difficulties, the greatest obstacle to program improvement stems from the manner in which the implementation system functions during program execution. The descriptive solution system specifies a problem by defining a series of relations (among, for example, input, output, sub-problem solutions, inferential systems, and solution goals). Within the execution system, the same elements have been reformulated in terms of the ways in which the relations are computed. In providing a logical/physical interface between these systems, the implementation establishes a general paradigm or framework within which the solution is expressed. Since this paradigm effectively creates an environment guiding processing activities, it will be referred to as the processing environment.

In this chapter, two general processing environments, sequential and applicative, will be contrasted. As their names suggest, the most obvious distinction between the two is the notion of program control; this is not, however, the only point of difference. Equally clear distinctions can be drawn on the basis of such features as the relative importance of data definition versus data manipulation in describing the implementation, the meaning of program symbols, the moment at which properties are bound to them, and the number and type of interfaces which are layered to form the implementation system.

A sequential processing environment views the underlying architecture as a traditional von Neumann machine. This does not necessarily mean that the processor controlling execution is sequential, but simply that the problem solution is described in terms of a sequence of steps to be carried out in a determined order. Historically, most views of computation have been based on the notion of instruction sequencing and by far the majority of existing systems operate along these lines. As a natural consequence, most software implementations make use of this processing environment.

The nature of sequential processing presupposes a relatively constrained representation of the problem solution describing its progression from start to finish; a program is therefore expressed in terms of a series of operations which manipulate data. For this reason, most programming languages designed for a sequential environment are called "procedural",¹ "algorithmic", or "imperative". They provide general data formatting capabilities as well as high-level versions of the elementary control sequences available on von Neumann architectures, such as repetition, selection, branching, and intersequencing (subprogram linkage). Although the sections on sequential processing occasionally make reference to specific language implementations, this is for illustrative purposes only. The similarities among procedural languages and sequential implementations

¹ The use of "procedure" in reference to a sub-program unit thus derives from the term "procedural" (i.e., sequential) describing the nature of program specification.

are so fundamental that they may be viewed as essentially homogeneous in terms of their implications for program optimization.

Although most existing systems are sequential in nature, AI programs are typically implemented in an applicative or non-sequential processing environment. This type of environment does not view the underlying machine as a von Neumann model of computation and therefore is not suited to the same types of optimizations. Whereas the sequential environment approaches execution as a discrete series of algorithmic steps, non-sequential processing revolves around such concepts as reduction, resolution, and unification. Again, this does not imply anything about the nature of the processor itself. In fact, since the majority of existing computer systems rely on sequential processors, most applicative processing environments are superimposed on sequential implementation layers. It is only with the growing interest in largescale AI programs during the past decade that non-sequential architectures have become a viable processing alternative.

Non-sequential problem solutions are more concerned with defining underlying relations than with prescribing their computation. Just as the procedural languages provide a natural expression of the sequential approach, non-sequential processing is best described by the "symbolic" or "definitional" languages. These can be subdivided into three groups which have evolved along divergent lines since the first symbolic language, LISP, was developed by McCarthy in the 1960s.

The largest and best known group encompasses "functional" lan-

guages, which view a program's output as a function (in the mathematical sense) of the input. During execution, successive reductions are used to simplify the program function until further applications are impossible. Recursion and functional composition are the primary control mechanisms, with each operation performed when the result it generates is needed by an invoking instruction.

In contrast, "logical" languages employ resolution and unification as their primary processing mechanisms. Program execution is approached in terms of proof derivation. A series of propositional logic implications formally describe the underlying terms or assumptions and any inferential relations between them, with the desired output expressed as one or more queries. To satisfy the goal, appropriate patterns are selected from the rule base to produce a solution space.

The third class includes the newest addition to the spectrum of programming languages, the "dataflow" languages. As the name suggests, these approach execution as inherently concurrent, with the firing of each instruction dependent solely on the availability of its data inputs. Dataflow operations, like their functional and logical counterparts, are expressed in terms of functional applications. Although recursion is eliminated as a programming tool, the mathematical notion of function is extended to allow the return of more than one value.

Applicative processing is not the only alternative to the sequential environment. Strictly speaking, the term applicative is

appropriate only with respect to a functional or reduction paradigm; unification and dataflow systems more properly constitute separate non-sequential entities. The only systems which are currently capable of meeting real-time performance standards, however, are applicative (this topic is discussed in more detail in the next chapter). Furthermore, the three systems share almost as many fundamental similarities as do the procedural languages -- as witnessed by the fact that virtually all post-experimental implementations of logical and dataflow languages are interpreted representations relying on applicative evaluators. In keeping with the objective of assessing the feasibility of optimization using available methodologies, therefore, the discussions of non-sequential processing emphasize the applicative model.

The sections which follow examine the functional characteristics of the implementation system in general before focussing on features specific to the two processing environments. Particular attention is given to those features which have implications for program optimization.

3.1 Structure of the Implementation Solution System

Between the computer which the applications software user sees and the physical machine controlling execution are a series of abstract or virtual computers. Each level in the hierarchy represents a functionally distinct machine with a specific instruction set,

resource configuration, and implementation strategy. The implications for optimization are critical: program improvement at one level does not necessarily result in efficiency at the next.

Figure 6 illustrates the abstract computer hierarchy for a typical implementation system. At the highest level is the machine defined by the applications program. The "program" it executes is the input data, and execution is expressed in terms of the operations available in the high-level language of the source program. Although the applications programmer does not normally conceive of his program as a "translator", in fact it functions as such by transforming the input data into a "target program" which is executed by the computer represented at the next level.

The operations of the applications program in turn serve as "input" to the virtual machine defined by the high-level language translator. This machine's actions are described by another instruction set, generally the operations that are expressed in mnemonic form as the assembly language of the target computer. The assembly-level instructions provide an interface to yet another abstract machine, this time executing the primitives provided by the operating system. At the lowest level of the implementation system -- just above the physical interface to the execution system -- is the microcode or sub-primitive abstract machine which translates operating system

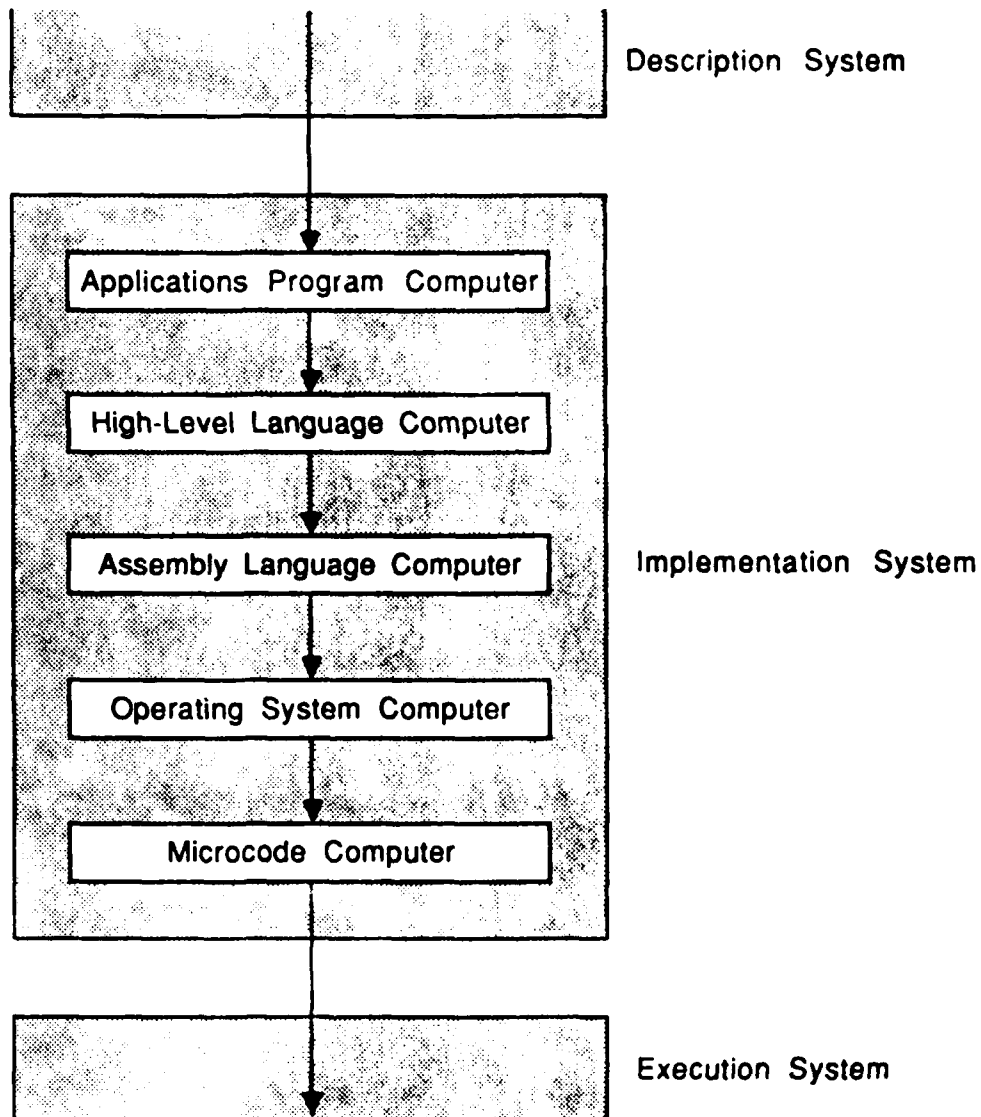


Figure 6. Abstract Machines in the Implementation Solution System

functions to physical machine instructions.¹

The abstract machine hierarchy thus bridges the semantic gap between the description and execution systems by successive interpretations. The multiplicity of layers greatly complicates optimization activities. At each level, operations must be expressed in terms of the available instruction set, generally without regard to the appropriateness with which those instructions will be translated into the next machine's operations. Inter-level transformations thus tend to be "black boxes" from which little can be assumed about the ultimate fate of optimization attempts.

The hierarchy of abstract machines clearly parallels the common stratification of programming languages into classes of varying degrees of machine dependence (Figure 7). The similarity may be misleading, however, since the selection of a particular language does not necessarily increase or diminish the number of implementation mappings. The level of abstraction of the language does determine to some extent the degree of optimization feasible. A programmer using assembly language, for example, can modify his solution to take advantage of specialized instructions, while the user of a very-high level language may unwittingly express the program in such a way that no improvement is possible. In general, the difficulty of trans-

¹ Many implementations actually involve more levels than are shown in Figure 6. The "operating system computer", for example, is generally sub-stratified into a library program level, a utility level, and perhaps a supervisory level. Each of these has a distinct set of primitives providing a transition to the next lower level.

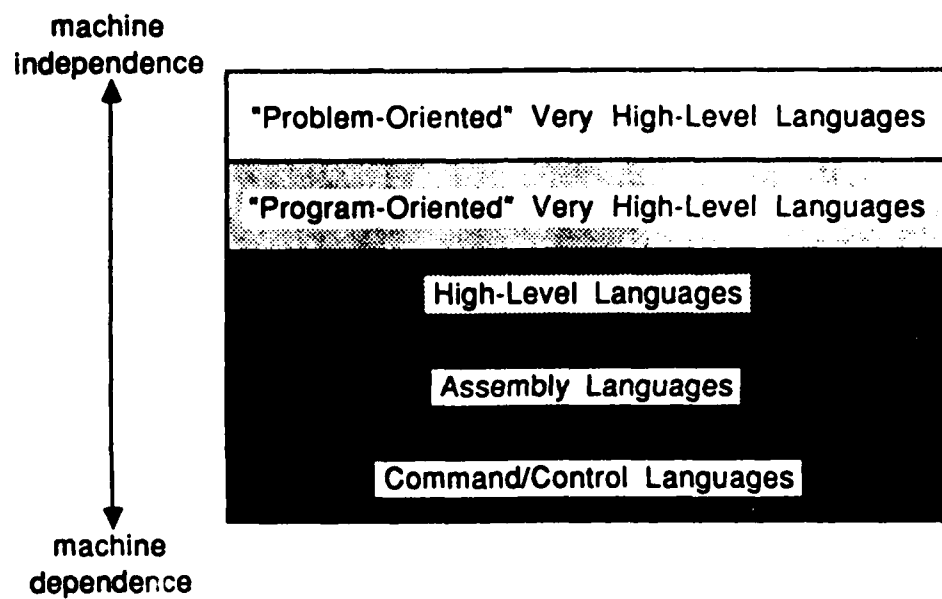


Figure 7. Programming Language "Hierarchy"

forming a program concisely and efficiently increases in direct proportion to the degree of machine independence. Subsequent chapters distinguish among levels of language where appropriate; if no mention of this is made, it should be assumed that the effect is negligible.

Another factor which imposes limitations on the nature and degree of optimization activities is the manner in which each abstract machine effects the transition from input to target program. Compilation systems statically analyze the input program and generate an output program which is completely expressed in terms of target instructions.¹ This output is ready for "execution" by the next level's machine as though it had originally been coded in that form; it is immaterial whether the program is executed immediately or at some later date. In contrast, interpretation systems defer most transformations until run-time, when the abstract machine interprets or simulates the input program through the use of library routines, which provide the low-level algorithms necessary to carry out each instruction of the higher-level language. Most implementation hierarchies include a combination of compiling and interpreting translators. This complicates the optimization process since transformations which are possible or desirable in one system may not be feasible in the other.

Thus far, the functioning of each abstract computer has been

¹ The process is also referred to as "translation"; in the present discussion, however, this term is used generically to include both compilation and interpretation.

characterized in terms of its interaction with adjacent machines. In the sections which follow, the implementation system is approached in terms of the framework imposed by the processing environment selected for program execution. It is important to note that the processing environment, like other features of the implementation system, is neither wholly physical nor wholly logical. It must concretize both problem definitions and computational procedures, yet it does so in terms of an abstract rather than an absolute view of the underlying physical architecture.

3.2 Characteristics of Sequential Processing

In the sequential environment a clear distinction is made between control and data elements, with a separate portion of program storage allocated to each. The control segment represents the series of instructions to be performed, so elements may only take on certain configurations (i.e., instruction codes accepted by the target abstract computer) and their ordering is crucial. With few exceptions, the control segment is immutable, that is, its contents are not altered during execution. The data segment provides storage elements which can be manipulated by the instructions. In this case the ordering and configuration of individual elements is incidental (although possibly subject to formatting constraints imposed by the target) and the code is mutable, or may be freely altered during execution. The control/data dichotomy is maintained throughout the

abstract machine hierarchy of the implementation system. The transformation effected by each level includes high to low level, one-to-many mappings of both instructions and data items. As we shall see, this characteristic division has important repercussions on the nature of optimization activities.

Figure 8 illustrates the layering of abstract machines typically found in sequential processing environments. Below the applications program computer lie two levels of system support implemented by means of software and firmware. Software support machines include those expressed in terms of the primitive operations provided by the programming language, library routines established on levels specific to or independent of the programming language, and the underlying subprimitives available through the operating system. (Ada language implementations are somewhat different from this configuration, since the KAPSE typically provides machine dependent, low-level services which wholly or partially replace access to the normal operating system subprimitives. In this respect, it is similar to the layering of non-sequential environments established on sequential architectures; see Figure 9.) The firmware level machine transforms the software instructions into the microcode subprimitives used by the hardware control unit.

The structuring of the programming language implementation may allow the applications program to directly access the library routines, but this is not always the case. These potential interfaces are indicated in Figure 8 by vertical channels bypassing other inter-

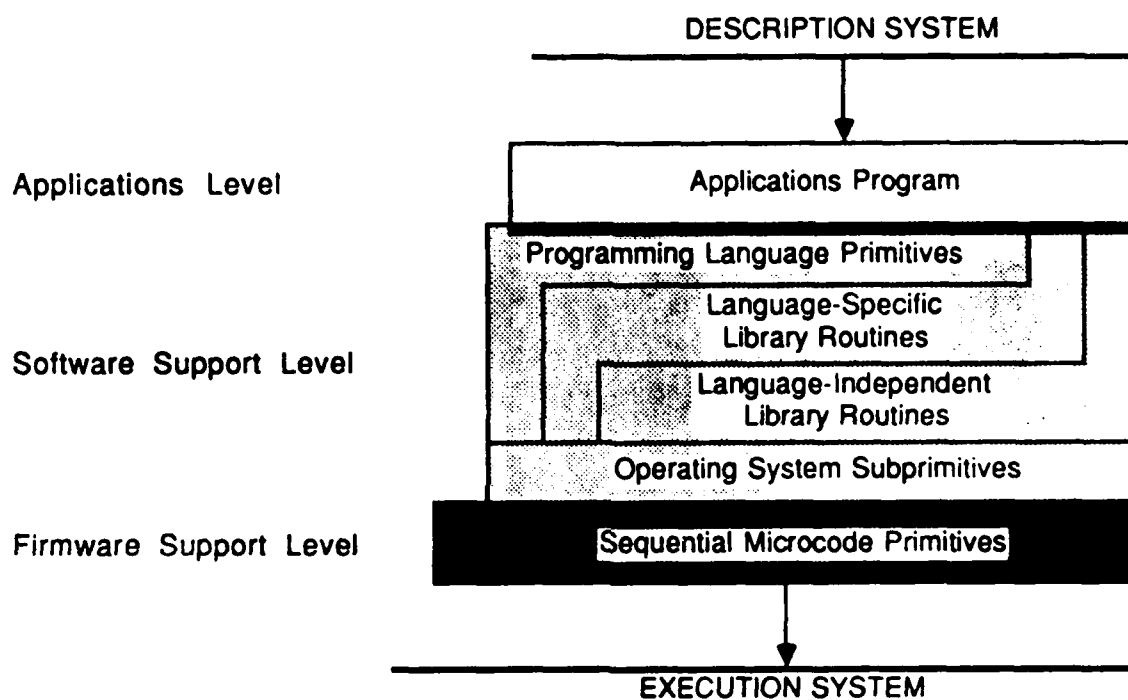


Figure 8. System Support Layering in Sequential Processing Environments

face levels. The implications of support layering should be obvious, since each level conceptually represents an abstract machine: the number of transformations required during execution is in direct proportion to the depth of support layers.

Suppose that a given machine M , in the system receives as input a version P_M of the program. P_M includes two portions, a control segment constructed from the instruction set for M and a data segment consisting of a series of storage elements; these are represented as $I_M = \{i_1, i_2, \dots, i_k\}$ and $D_M = \{d_1, d_2, \dots, d_k\}$, respectively. M transforms P_M to an output version $P_{M'}$, targetted to the next machine, M' -- i.e., $I_{M'} = \{i'_1, i'_2, \dots, i'_m\}$ and $D_{M'} = \{d'_1, d'_2, \dots, d'_n\}$. Because M' is closer to the physical level than is M , it is typically the case that $|I_{M'}| > |I_M|$ and $|D_{M'}| \geq |D_M|$. Below M' are additional machines M'' , etc., each processing a more primitive level of instructions.

Since it represents the cumulative efficiency of all abstract machines in the system, program efficiency is in general adversely affected by the number of support layers. When a short-circuit channel exists from M to M'' , however, it becomes feasible for the output program to include some instructions which are already targetted to the lower machine, thus obviating one or more levels of transformation. A well designed translator can mitigate the effects of layering by expressing the output program in such a way that it can continue to be transformed efficiently at lower levels and/or take advantage of short-circuit channels to bypass processing.

3.3 Optimization in the Sequential Environment

To optimize program storage space, it is clearly necessary to minimize

$$\text{size}(P_{M^n}) = \text{size}(I_{M^n}) + \text{size}(D_{M^n}),$$

where M^n is the final (lowest) abstract machine in the implementation system. (Note that a minimal total transformation does not necessarily mean that each partial transformation is minimal, although this is typically the case.) Time optimization is not as easy to characterize. The generalized view of execution (employed in the preceding chapter) as a linear series of program actions must now be reformulated in terms of the abstract machine instructions used to express the program, including nonlinear control directives. Total execution speed will depend on the speed of each target instruction and the number of times -- possibly none -- it is executed, rather than the number of instructions (i.e., $|I_{M^n}|$).

The remainder of this section will refer to program execution in terms of the operational semantics of a single abstract machine, M , selected arbitrarily from the implementation system. In describing the effects of executing the program P_M , it should be obvious that we are modeling the simultaneous operation of all abstract computers in the configuration. This is consistent with the observation that available optimization techniques are based on general aspects of the control/data dichotomy rather than features specific to any single

processing level.

Program execution in the sequential processing environment is commonly viewed as a series of transitions from one program state to another. Each state is described in terms of a pointer to the next operation in the control segment and the current configuration of the data segment. In terms of the notation already established, a program state is therefore a pair (i, D) , where i is the instruction to be executed next and D the contents of the data segment. Execution is modeled as a computation sequence, or series of the form

$$(i_0, D_0), (i_1, D_1), \dots,$$

indicating the progression beginning at the initial program state. For each state in the sequence, the successor state is determined on the following bases:

$$i_{n+1} = \begin{cases} \text{next_instruction}(i_n) & \text{if } i_n \text{ does not cause a branch} \\ \text{target_instruction}(i_n) & \text{otherwise} \end{cases}$$

$$D_{n+1} = T_n(D_n),$$

where T_n is the transformation on data elements effected by the instruction i_n . If the program ends, the series is terminated by a final state; otherwise it cycles indefinitely. A snapshot is a state pair (i_j, D_j) in the computation sequence, representing the configuration after $j-1$ statements have executed. (Note that j refers to a position in the chronology of execution and not to a relative location in the program definition. It is therefore possible that the control segment (I_M) might include instructions which are not attainable from

the initial state. Any instruction occurring in a program state snapshot, however, is by definition reachable through at least one computation sequence.)

It is important to observe that the computation of the next instruction of the control segment is distinct from any transformation made to the data segment. Because to a large extent control and data function autonomously (with the obvious exception that transfers of control may be contingent upon data values), they may be analyzed independently; hence the terms control flow analysis and data flow analysis. Control flow analysis establishes the feasible progressions of the I components of program states by considering what instructions may be executed in what sequences beginning at the initial state. Data flow analysis, on the other hand, concentrates on the range of the D components by determining what values may be taken on by individual data items.

Appendix A describes optimization activities in the sequential environment and summarizes the results of studies attempting to quantify what effect optimizing transformations have on program performance. In spite of the inconclusive nature of this data, it is possible to generally identify the features of the sequential processing environment having greatest influence -- favorable or adverse -- on optimization potential.

The importance of the support system configuration has already been described. Both the number of layers and the quality of the translations influence the overall effectiveness of improvement

efforts. In addition, optimization activities are favorably influenced by such programming practices as the intelligent selection of data formats to minimize the need for coercion or conversions, the organization of expressions to facilitate the application of algebraic transformations, and a careful placement of non-optimizable subexpressions (such as those involving invocations of user-defined functions) with respect to loops or other control structures. Optimization is adversely affected by the use of language features which interfere with control and data flow analysis or value propagation. These include, for example, aliased or dynamically allocated variables, global or other data storage which is modified by side effects, unconditional transfers spanning several control structures (e.g., a GOTO whose target is outside the boundaries of the enclosing logical interval), the use of control variables (i.e., entry and label variables) and/or directly or indirectly recursive subprogram units, and mixed mode expressions.

Optimization techniques in the sequential environment, as we have seen, rely extensively on a few fundamental assumptions: (1) a bipartite control/data organization, consisting of an immutable control segment and a mutable data store; (2) the use of program symbols as references to specific locations in the segments; (3) a subsequent dependence on static (pre-execution) analysis to associate or bind symbols to locations; (4) an ultimate restriction of program control to very simple primitives, namely sequencing and conditional branching; and (5) a corresponding reliance on iteration and selection

as the basic control flow mechanisms. Any deviation from these can seriously impede improvement activities at all levels. Consequently, the two factors most important in establishing potential for optimization are, without doubt, the way in which the problem solution is expressed by the programmer¹ and the configuration of support layers in the implementation system.

3.4 Characteristics of Applicative Processing

Unlike the sequential environment, applicative processing does not employ a bipartite organization nor does it approach problem solution in terms of a series of operations manipulating data elements. Instead, a program is a function applied to the input; the resulting value is the output. Since no real distinction is made between program data and problem data, a single common representation is used to represent all program elements internally. Function definitions and data items are both stored as linked lists composed ultimately of atoms. Even language primitives are atoms like any others, distinguished only by the fact that they are defined by the system when execution begins.

Furthermore, the value of a symbolic language function, like its mathematical counterpart, is determined solely by the values of

¹ Recall that this is independent of any consideration of how well or poorly the selected algorithm is suited to the problem, although of course even the best optimizer cannot compensate for an inefficient algorithm.

its arguments. This property, called referential transparency, has a profound impact on processing organization. At a given point during execution, the computation to be performed depends only on current context, not on the history of actions which led up to it. The notions of program state and computation sequence are therefore entirely absent from the applicative environment.

Since program symbols no longer represent locations in the data store, the fundamental units of procedural language programs (expressions and assignments) lose their power in the applicative setting. For example, the instruction

$$A = B$$

in the sequential environment indicates that the contents of location B should be retrieved and copied to location A, overwriting any previous value. In the applicative environment, an instruction of this sort is viewed as establishing a definition or association between the values of A and B, rather than performing an operation. Since A and B have no "locations" per se, values are not directly stored or copied, a trait reflected in language syntax, which restricts the appearance of a variable to the lefthand side of only one equation per program. (Note that because it is the notion of definition rather than computation which is implicit, the statement $A=A+1$ is meaningless in the applicative environment.)

Instead, the power of the applicative environment derives from the homogeneous treatment accorded to all program elements. Programs can be created dynamically by other programs and then executed. A

data list may be transformed into a program list and vice versa, with few if any restrictions. Since program symbols are viewed as values, they may freely utilize non-consecutive bounds or properties which are established and altered dynamically.

In keeping with the notion of functional organization, the primary control mechanisms are purely applicative: functional composition and recursion. Even the conditional construct is generally a simple variant of the projection function, whereby the value returned is the first one computable by the terms of the construct. Operations on data items are considered to have locality of effect, that is, they produce new items rather than altering existing ones. As we shall see, this has important implications for parallelism.

An immediate consequence of the functional approach to problem solution is that almost no von Neumann hardware-related features can be used directly by the applicative environment. If the underlying architecture is sequential, program execution must be simulated by one or more extra abstract computers that indirectly interpret applicative actions, using sequential software routines which can be translated directly into machine primitives. Figure 9 illustrates the configuration of system support layers typical to most symbolic language implementations. Figure 10 represents the configuration when the underlying architecture is non-sequential. In this case, much of the software simulation can be replaced by a firmware interpreter that translates software primitives directly.

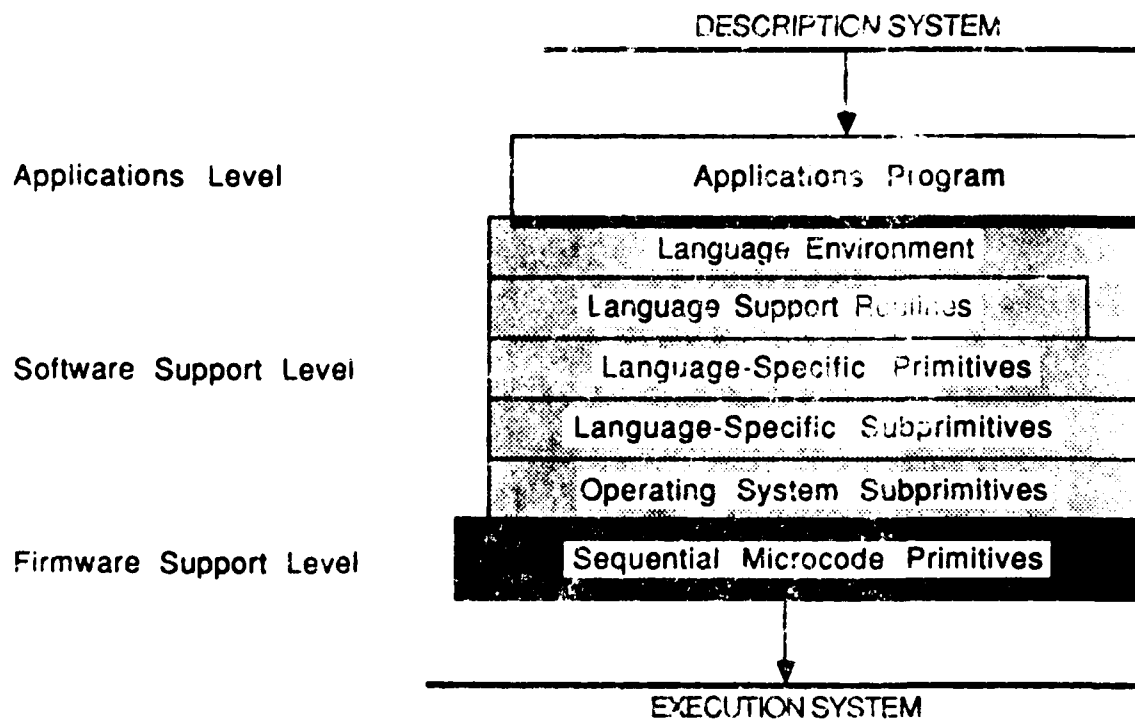


Figure 9. System Support Layering in Applicative Environments Hosted on Sequential Processors

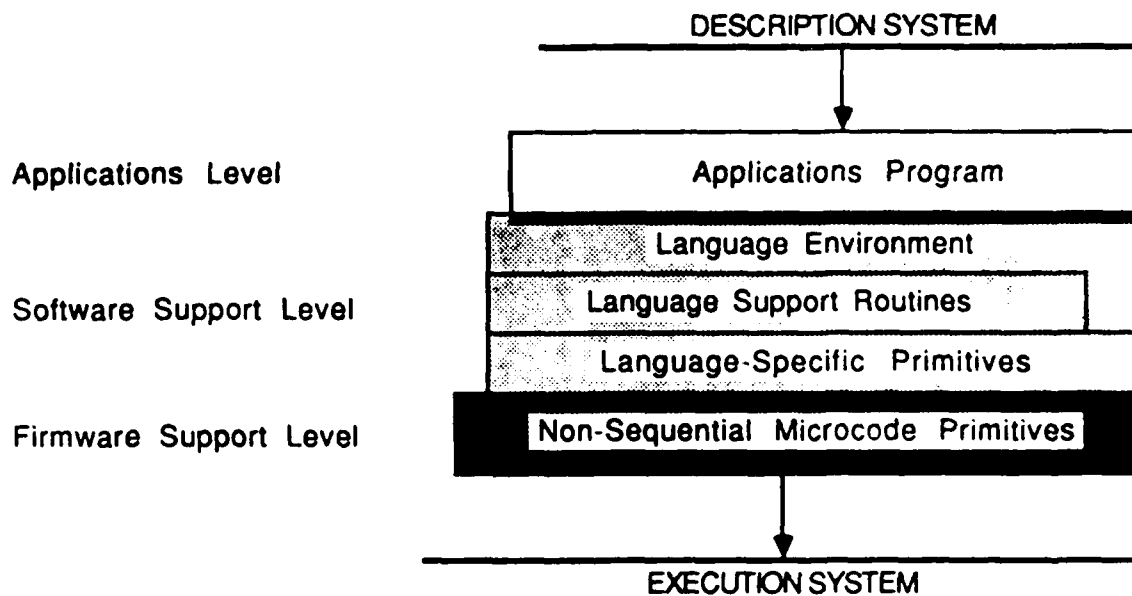


Figure 10. System Support Layering in Applicative Environments Hosted on Non-Sequential Processors

Below the applications program machine is an integrated language environment, which supplies debugging aids and perhaps other development support tools. Its output is in the form of the primitives used by general language support facilities, primarily the interpreter and the garbage collector. At the next layer, each processing primitive is represented by a software routine which simulates the basic operations such as functional invocations, data referencing, and storage allocation and deallocation. It is at this point that the distinction between sequential and non-sequential hosts becomes apparent. In most configurations, the central control mechanisms must be simulated by means of a software subroutine which effectively converts execution from applicative to sequential form; from this point on, the abstract machines will be identical to those of the sequential environment depicted in Figure 8. If, instead, the target execution system employs a symbolic processor, the primitives are translated via microcode at the firmware support level.

It should be clear that a primary cause of inefficiency in most existing applicative environments is the excessive amount of simulation required. The configuration of Figure 9 suffers not only because of the inordinate number of transformations which must be applied, but also from the difficulties inherent in interpreting non-sequential instructions in sequential form. The situation is further exacerbated by the exigencies of late binding; even primitive arithmetic operations must be "simulated" to the extent of performing run-time type checks and conversions.

3.5 Optimization in the Applicative Environment

Since the concept of program state is missing in the applicative environment, optimizations based on normal control flow and data flow analysis are inappropriate. Unfortunately, few optimizing transformations have been developed specifically for the applicative environment. This can be blamed to a great extent on lack of demand, due partly to the lack of popularity of symbolic programming prior to the mid 1970s and partly to the habitually casual approach to efficiency taken by the artificial intelligence community as a whole. It is to be hoped that the recent interest in real-time AI applications will result in new developments in this area.

The simplest way to achieve significant performance improvements is to take advantage of the compilation facilities offered by many applicative implementations. A compiled function is transformed into instructions corresponding to those produced by the language primitive abstract computer. When the function is invoked, only the firmware interpreter (or its multilayer software equivalent in the case of a sequential host) is needed to complete the translation process.¹ Although this cannot be considered an optimization technique in the strictest sense, it is the only means of improvement available in many

¹ It is not often possible to entirely eliminate high-level interpretation, however, since most configurations require that even compiled functions be activated through the language environment layer rather than by direct user invocation.

systems.

Like their sequential counterparts, symbolic language optimizers make several assumptions about the nature of input programs; deviation from these norms impedes or precludes improvement. On the basis of locality of effect, data dependencies are considered to be localized, i.e., subprogram units have no side effects. Since the basic control mechanisms are functional composition and recursion, programs are assumed to be made up of a large number of small, often recursive units. This means that a substantial portion of execution time is devoted to activating the linkages between units. Finally, late binding is quintessential in the applicative environment. Static analysis cannot suffice to associate symbols with attributes since the properties of both functions and data items may be created, redefined, or destroyed at arbitrary points during execution. Instead, a heap area must be maintained in which storage may be allocated and deallocated in a relatively unstructured fashion.

Program execution in the applicative environment can perhaps best be seen as the alternation of two activities, substitution and simplification. Substitution (unfolding) refers to the action of replacing a program symbol by its definition. This is followed by simplification (evaluation), which replaces the definition by the result obtained through evaluating the body of the definition. In the case of atoms and data lists, simplification is trivial, since the value is obtained through a search of the storage area. For functions, simplification will require the application of additional

substitutions and simplifications, perhaps recursively.

Appendix B describes the optimization techniques which have been developed for applicative environments. An attempt is also made to evaluate the results of studies on the effects of applying such transformations. It is quite difficult to characterize the features of the applicative processing environment having greatest influence on optimization potential, other than the presence/absence of the applicative-to-sequential transformation. The paucity of behavioral data, coupled with the disturbing nature of available results, relegate such efforts to pure speculation.

If we assume that current techniques address the real problems -- and this is by no means a safe assumption -- then improvement should be facilitated by the concentration of numerical computations in fewer subprogram units and a reliance on purely applicative constructs. The use of "special" features which make a symbolic language resemble a procedural one should be avoided, particularly iterative structures, GOTO-like transfers, and pathological binding strategies. The use of type declarations, however, should be beneficial since it would facilitate several types of improvements.

Optimization techniques in the applicative environment rely on the following assumptions: (1) a homogeneous internal representation; (2) the treatment of program elements as definitional values rather than storage locations; (3) a subsequent dependence on dynamic binding; (4) the use of functional composition and recursion as the primary control mechanisms; and (5) a corresponding reliance on the

properties of referential transparency and locality of effect. The elimination of system support levels by moving implementations to non-sequential processors is clearly the best way of achieving significant performance improvement given current methodology. The application of automatic optimizations must be viewed with some skepticism until their effectiveness is demonstrated by empirical study.

4 A Strategy for Maximizing Optimization Potential

Previous chapters discussed the nature of program performance and the ways in which it can be enhanced in typical implementation settings. As we have seen, the types of improvements that may be applied are determined by the characteristics of the processing environment. Particular optimizers may utilize larger or smaller subsets of the available techniques with greater or lesser effectiveness, but the limits are established globally by the environment itself. Since our concern is with the efficiency of AI algorithms in the implementation solution system, it follows that we should focus our attention on the selection of the processing environment.

The ideal environment would be one which guarantees optimal program performance in all cases. Since efficiency and optimization are at best relative terms, this is patently impossible. It is unlikely that any single configuration can predictably maximize the performance of even a small subset of the AI problems posed by BM/C³ applications. How, then, are we to realistically select an environment for the optimization of an arbitrary AI algorithm? The sections which follow establish criteria for evaluating processing environments in terms of their responsiveness to program needs. A divide-and-conquer form of implementation is then presented. This strategy partitions a program into segments for processing in a heterogeneous environment, thereby maximizing program "optimizability".

4.1 Evaluating Processing Environments

Any process which relies on successive transformations falls prey to the dangers of inefficiency and inaccuracy. The implementation system, responsible for bridging the quite considerable gap between conceptual and physical solution, is undoubtedly a major source of performance degradation in all computing systems. The development of efficient AI programs must ultimately depend on the capability of the processing environment to apply automatic improvements which at least partially compensate for this introduced inefficiency.

One criterion for choosing a processing environment is the number of abstract machines in the system. The relationship of system support layering to performance has already been addressed. Clearly, the best layering configuration is that which will require the fewest number of translations in executing the program. Relevant considerations include the number of layers present, the availability of short-circuits to bypass intermediate levels, and the fact that individual layers do not necessarily play equal roles in determining the overall effectiveness of the system. An alternative is to select optimizers which exploit the nature of a particular layering configuration. Since optimizing transformations are never applied uniformly across all portions of a program, this involves assessing the relative likelihood of preconditions for improvement and the ability of translating mechanisms to take advantage of short-circuit channels, as well as the effectiveness of each type of transformation. It is obviously

desirable that improvement efforts be applied at all levels of the implementation to achieve optimum performance. The two approaches can be combined in an approximation of the minimax game-playing heuristic. The goal is to choose the system configuration which combines a minimum number of abstract machines and a maximum degree of automatic improvement. Such a selection sets an upper bound on the degree of efficiency attainable by an arbitrary program, but it cannot guarantee a lower bound.

Another aspect of the processing environment is that it provides a computational paradigm within which the problem solution must be structured. Here we find that the sequential and applicative environments, like the languages which naturally express each paradigm, differ radically. The procedural/sequential approach to problem solution concentrates on data manipulation and alteration through a strictly ordered sequence of operations. The functional/applicative solution, on the other hand, computes by value rather than by effect, so the program description focuses on relationships instead of the ways in which they are computed.

In terms of computational power, the two paradigms are approximately equal. Sequential processing corresponds to the Turing model of computation with its clear delineation of control and data. Program instructions are encoded in an immutable store and selected for execution by means of simple sequencing or conditional transfers; operations can examine or alter the contents of the mutable data store. Associated with each action is a program state, which encaps-

ulates the history of the computation to that point and determines the next action to be taken. Functional processing approaches computing from the standpoint of recursion theory. In this case, control and data elements are treated homogeneously as values. Execution is a process of functional evaluation, sequenced by functional composition and recursion; the next computation is thus determined by context rather than history. The class of problems computable by means of recursion theory is not, strictly speaking, as general as that described by the Turing model. Since the differences are pathological, however, we can view the two as equivalent for the implementation of AI algorithms.

The types of optimization appropriate to each environment have been discussed in considerable detail. Unfortunately, it is impossible to compare the two impartially in terms of performance since published findings are vague and self-contradictory. Figure 11 illustrates a single benchmark observed by [Gabriel 85] on a variety of systems. The results are totally inconclusive. Few details are explained in the report (e.g., details of comparative machine configurations and some of the options are not described) and the experimental conditions do not survive close scrutiny. Furthermore, the Tak benchmark itself is of questionable utility since it involves some 64,000 recursive calls and 48,000 decrements but nothing more.¹

¹ That Gabriel was not purporting to compare LISP to other languages is immaterial: the findings for individual LISP systems are subject to the same lack of coherence.

Machine	Language	Options	Timing
VAX 11/750	Franz Lisp	generic arithmetic	19.9
		generic arithmetic	11.6
	PSL*	generic arithmetic	7.1
	Franz Lisp	integer arithmetic	3.6
	C		2.4
	Franz Lisp	integer arithmetic, fast function call	1.9
	PSL	integer arithmetic	1.4
VAX 11/780 (Diablo)	Franz Lisp	integer arithmetic	2.1
	C		1.35
MC68000	Pascal		3.8
	PSL SYSLISP		2.93
	C		1.9
	Machine Language		0.7
Stanford AI Laboratory	MacLisp		0.832
	MacLisp	declarations used	0.564
	Machine Language	"wholine"	0.255
	Machine Language	"ebox"	0.184

* Portable Standard Lisp

Figure 11. Gabriel's Tak Benchmark
Comparing Lisp with Procedural Languages

Statistically relevant data comparing the two environments is simply not available at any level.

The expressive power afforded by the two environments is also difficult to compare, primarily because the computational paradigms are essentially incomparable. In general, the symbolic languages allow a simpler, more elegant description of problems. Output can be clearly expressed as a function of input and a single algorithm can be uniformly applied to individual data objects or entire classes of them. Procedural languages, on the other hand, are more natural to most programmers. Their form is familiar and reflects the human tendency to view problem solving as a sequence of actions. The selection of an environment on the basis of this criterion must ultimately depend on the problem to be solved. Some problems are inherently sequential and others inherently recursive; it is as difficult to express the former in terms of an applicative solution as it is to describe the latter sequentially.

Representational power describes the suitability of the environment for implementing AI programs. The general objective of artificial intelligence is to encode knowledge about some domain and then use that knowledge to solve problems in the domain. The environment selected must provide suitable means for encoding information, retrieving data that is relevant to the problem, and determining a satisfactory solution. Furthermore, the system must conform to software engineering standards for verification, maintenance, and so forth. The procedural languages are criticized for

their emphasis on calculations rather than fundamental relationships and the inherent difficulty of proving program correctness. The major complaints against symbolic languages are the convolutions necessary to express simple sequencing, the lack of applicability of standard testing techniques, and the intrinsic inefficiency of heap storage management. Overall, the emphasis given to problem relations makes the applicative environment somewhat more appropriate for representing typical AI problems. The sequential environment, on the other hand, is better suited to the application of software engineering methodology.

In summary, neither the sequential nor the applicative environment possesses an undisputed superiority for the efficient implementation of AI algorithms. Each approach has inherent strengths and weaknesses which can have significant impact on the performance of largescale software systems.

4.2 The Environment Spanning Strategy

The AI algorithms needed for the BM/C³ setting can be categorized in general terms as search, reasoning, or constraint satisfaction problems. A search algorithm conceptually views the entire solution space and attempts to find a suitable path through it (e.g., track discrimination problems). A reasoning algorithm accumulates data by deducing it from previous truths and adds it to the knowledge base for future deductions (e.g., attack assessment

expert systems). A constraint satisfaction algorithm successively eliminates portions of a rule base that are inconsistent with the constraints imposed by the goal to ultimately derive a set of objects which satisfy the conditions (e.g., resource allocation problems). Large AI systems often combine more than one type of algorithm to solve complex problems, applying a divide-and-conquer technique to reduce the magnitude of the problem to solvable proportions.

The lack of conclusive evidence concerning the superiority of one processing environment over the other in terms of the general needs of AI algorithms suggests that a divide-and-conquer strategy might be appropriate here too. To investigate this approach, we will outline a representative BM/C³ problem and a possible solution. Our goal in proposing an implementation strategy is to maximize what we will refer to as the program's optimizability. Intuitively, optimizability measures the extent to which a program can be improved by available optimizers. This is equivalent to assessing how well the problem solution typifies the preconditions for applying compatible optimizing transformations.

The case study is an extension of work originally performed by Optimization Technology, Inc., and later supplemented by Auburn University. The initial study [OTI 86] dealt with the feasibility of applying symbolic processing techniques to algorithms operating on optical sensor data in an SDI environment. The Forward Acquisition Sensor (FAS) algorithms developed in Pascal by Nichols Research Corporation were used as the subject. The set was designed to perform

sensor measurement processing and precision track and discrimination functions; it includes color correlation, scan-to-scan correlation, single target processing, discrimination, irradiance calibration, and stellar attitude update routines. The OTI study concentrated on the area of scan-to-scan correlation, where it was felt that the greatest improvements could be realized by reformulating the Nichols algorithms for applicative processing. Benchmark results were compared for versions in Pascal and LISP on a VAX 11/780. Pascal out-performed LISP in data storage tasks (in spite of the fact that there was some pro-LISP bias in the data structures chosen for the programs), while LISP was faster at windowing transformations. The OTI study considered these to be mixed results, an understandable reaction in view of their stated desire to demonstrate the superiority of symbolic processing in situations requiring the dynamic correlation and updating of large amounts of data.

From our viewpoint, however, this case typifies a problem common to most AI programs in real-time settings. Some of the subtasks involved, such as I/O, sorting, numerical calculation, and storage operations represent exactly those operations which are intrinsically suited to the sequential paradigm. Others, such as pattern matching and discrimination, intuitively fall into the realm of applicative processing. Each system performs well when most processing is of an appropriate type, and each can be overwhelmed when subjected to large amounts of unsuitable activity. Most programming languages appear to provide for both sequential and applicative activities by incorpo-

rating syntactic features mirroring those of their counterparts. Since the two paradigms are so radically different, however, the resemblances are strictly superficial.

To demonstrate the important role played by computational suitability in setting an upper bound on performance, researchers at Auburn University extended the OTI routines. First, it was concluded that the existing programs did not in fact particularly exercise those task areas for which LISP was most appropriate (e.g., uniform treatment of program and problem data, recursion, and properties requiring dynamic binding). The manipulation of dynamic property lists was added to the original program to correct this bias. In keeping with the nature of general discrimination activities, the property lists were designed to represent any of a variety of supplementary data gathered sporadically on a by-demand basis by special-purpose sensors. Since this information does not apply to all tracked objects and is updated only occasionally, it would create undesirable burdens on data processing if incorporated directly in the main data store; instead property lists are allocated dynamically when and if needed. Particular care was taken to make sure that the benchmarks were as equivalent as possible, given the syntactic and semantic limitations of the two languages.

The resulting differences in performance were as expected. LISP clearly out-performed Pascal when dynamic capabilities were required, but did poorly when a sequential paradigm was more appropriate. Track initiation is another FAS situation where symbolic processing should

emerge a clear winner since, as the OTI study pointed out, considerable savings in calculation could be realized if closely clustered objects can be treated as a single object for analytical purposes until such a time as their tracks diverge. On the other hand, computation intensive tasks such as discrimination and calibration would undoubtedly perform better as sequential implementations.

The solution is clear: real-time AI algorithms should be developed for an implementation system which allows a true combination of sequential and applicative processing. We therefore propose a strategy that utilizes available methodology to combine the two computational paradigms. Since an implementation of this type must clearly bridge two distinct environments, the strategy is referred to as environment spanning. Although it represents a somewhat radical departure from the typical implementation system, the description and execution systems remain unaltered and no special equipment or techniques are necessary to effect the change.

Environment spanning begins during the initial program design phase, when the implementation system is originally selected. The problem solution, rather than being expressed uniformly in terms of a single paradigm, is partitioned into groups of subtasks suited to sequential and applicative processing. The criteria for assigning an activity to an environment will normally be those established in previous sections, although they do not preclude the possibility of utilizing algorithms or modules which have already been implemented according to one paradigm or the other. The software components from

the two environments are ultimately interfaced using one of the environment spanning methods described below.

4.3 Environment Spanning Implementations

Environment spanning represents a type of heterogeneous computing environment, that is, a configuration in which dissimilar hardware and/or software systems function cooperatively. The exponential increase in available hardware and software components during the past two decades has inspired the development of a number of heterogeneous environments, but most efforts have been devoted to interfacing physical elements rather than software facilities. Since environment spanning affects only the implementation system, the actual hardware configuration is irrelevant. What is essential is some means of coordinating sequential and applicative processing components.

The key to exactly what type of cooperation is needed can be found in the nature of the structural frameworks imposed by the two paradigms. Symbolic programs are effectively limited to evaluation activities, which restricts the applicative/sequential interface to the passing of functional values. Since multivalued functions are not supported by the procedural paradigm, this is further restricted to the passing of single values. Clearly, what is needed in each of the two spanned environments is some means of remotely activating a component of the other and later receiving a returned value. There are three major ways of achieving this.

The most straightforward environment spanning method is the simultaneous operation of sequential and applicative systems in a parallel architecture. Figure 12 illustrates this type of configuration; the arrangement is equally suitable when the applicative environment is superimposed on a sequential processor. The components are developed independently on the two processors and eventually exchange signals and data via the communication channels established for the parallel system. At least two symbolic processor manufacturers are currently engaged in developing architectures which combine symbolic and sequential processors linked by a common bus, but no systems are commercially available at the present time. For this reason, although parallel processing is conceptually the clearest form of environment spanning, it cannot be considered consistent with our objective of staying within the confines of available methodology.

A second spanning mechanism is implicit in multiprocessing environments (for our purposes, it is immaterial whether the processes are in fact executing in parallel or the concurrence is simulated). Here, as depicted in Figure 13, separate tasks are created representing the sequential and applicative environments. Remote activation and the passing of functional values are handled through the semaphore/mailboxing facilities of the underlying system. Unfortunately, not all multiprocessing environments allow the spontaneous creation of applicative tasks. In these cases, the most likely alternative is to construct the applicative environment as though it were the only means of processing; then, using the

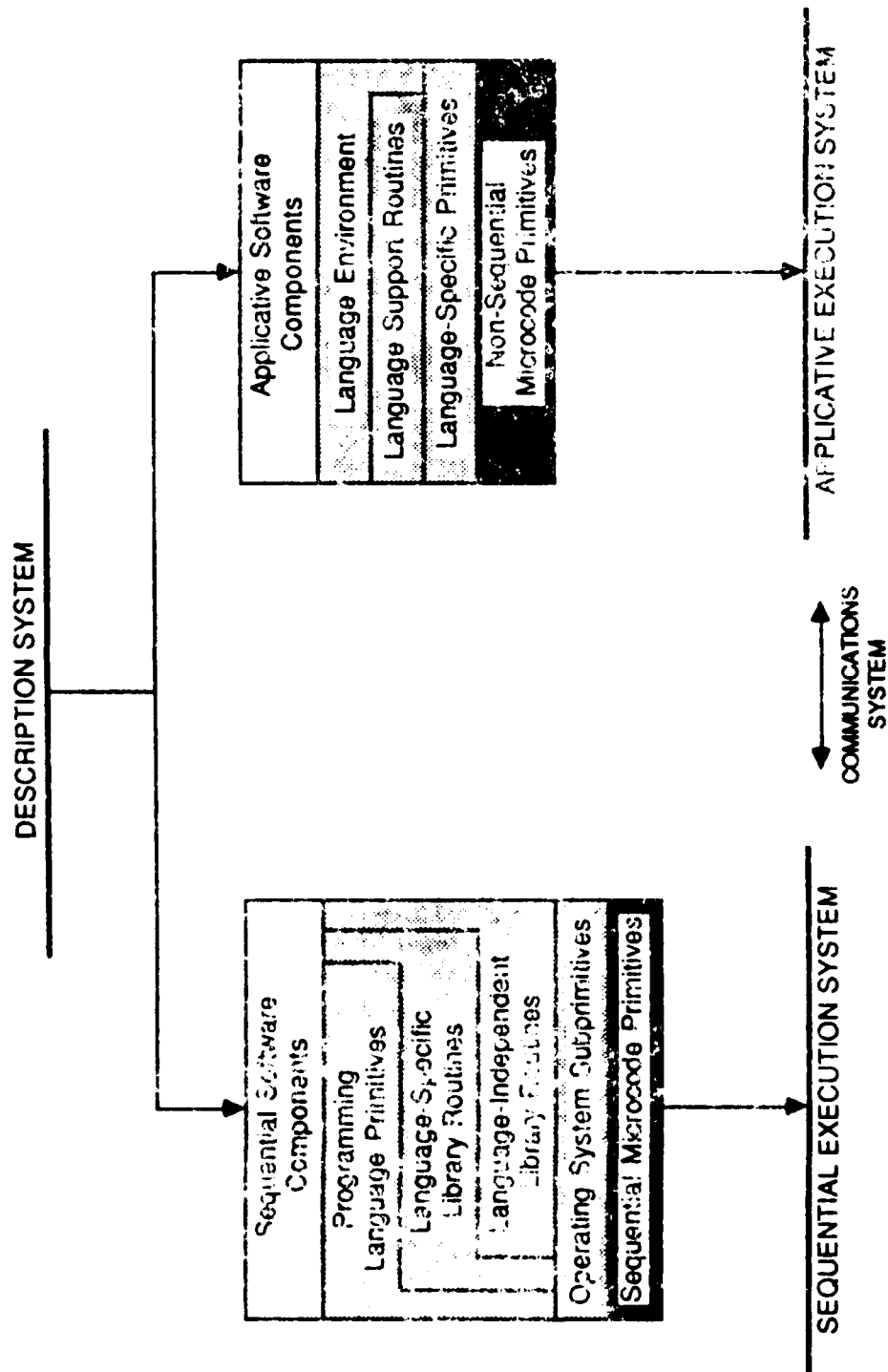


Figure 12. Environment Spanning Using Parallel Processors

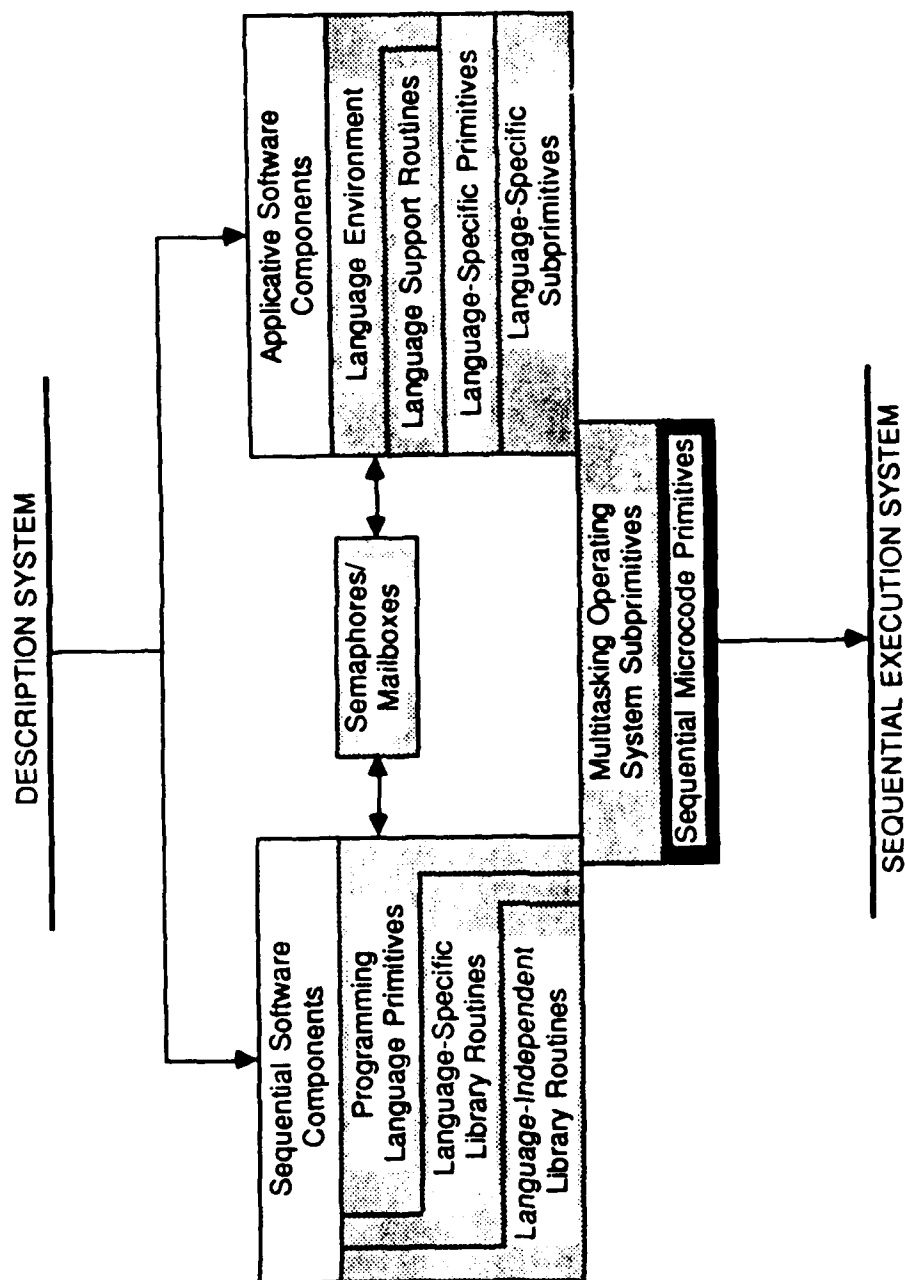


Figure 13. Environment Spanning Using Multitasked Processing

intersequencing method described next, invoke a sequential module which effectively spawns the second environment. This would allow concurrent processing on a simplified level.

The third method has no particular system requirements other than the ability for one environment to invoke modules created in the other; this is true of most operating systems which support both symbolic and procedural language implementations. Intersequenced environment spanning establishes a master processing environment which invokes elements of the slave environment when needed (see Figure 14). Restrictions on the interfacing between symbolic and procedural language modules in most operating systems require that the master be whichever component includes a language environment layer, normally the applicative. Under VAX/VMS, for example, a LISP program can directly invoke modules written in either symbolic or procedural languages, but no procedural language can invoke LISP modules since there is no way of establishing the needed LISP environment layer.

Which of the three solutions is selected obviously depends on the nature of the available hardware. The actual implementation method could be transparent to the software designer/implementor if an environment spanning interface were developed. This would automatically reformulate communicating components as needed to conform to the specific requirements of the spanning mechanism.

It should be noted that environment spanning does not impose any relative balance of processing on the two systems (although within an environment more typical multiprocessing may, of course, be going on).

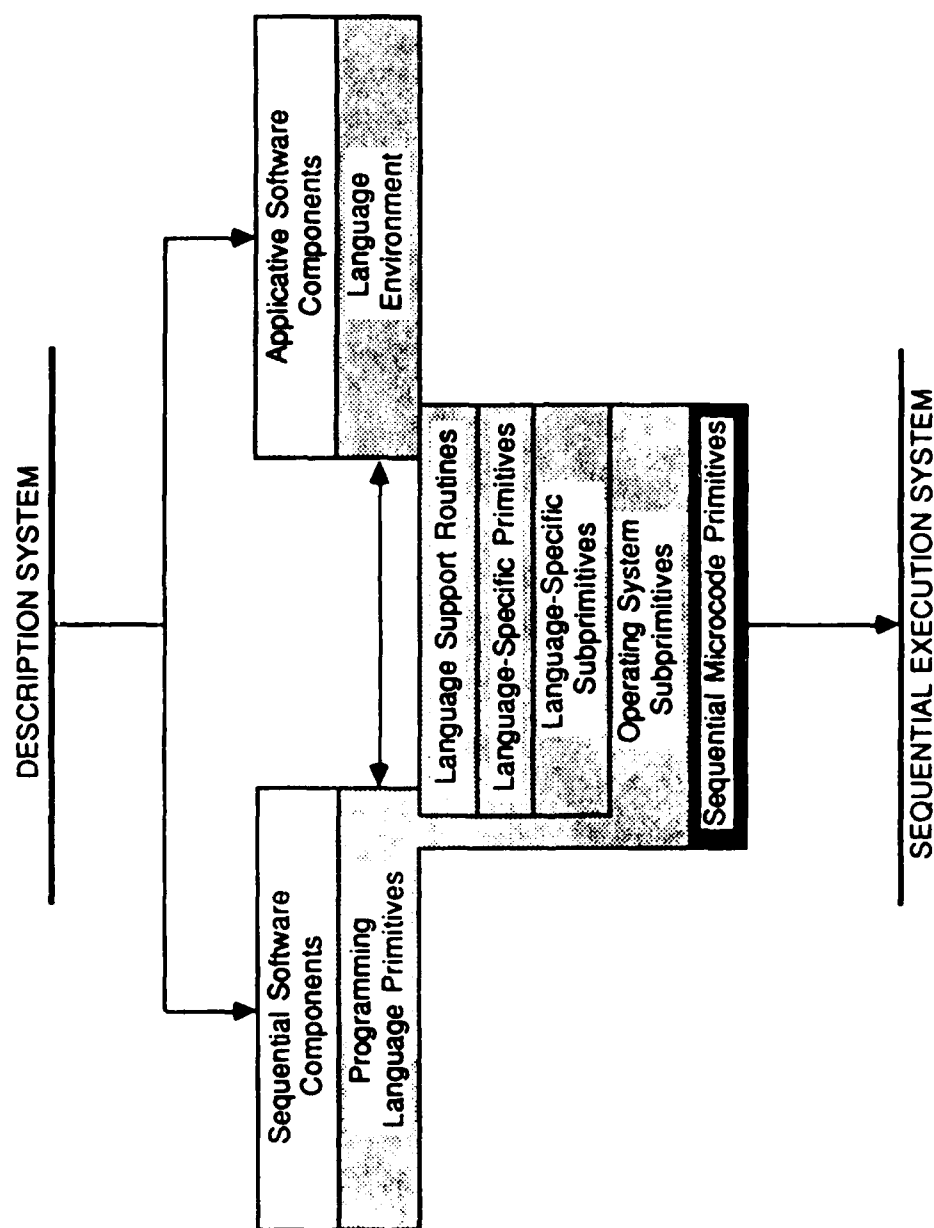


Figure 14. Environment Spanning Using Intersequenced Modules

The issue here is to partition program activities, not in order to distribute processing according to any particular pattern, but rather to allow the expression and ultimate execution of each task in the environment most appropriate to it. Optimization potential is limited by a combination of factors, each related to the computational paradigm chosen for the implementation. By realistically assessing the characteristics of each program component and assigning it to an appropriate processing environment, the designer/implementor maximizes the effectiveness of optimization efforts throughout the implementation solution system.

5 Conclusions

The stringent constraints imposed on computing systems in the BW/C³ setting have forced the issues of program performance on the artificial intelligence community. Can AI algorithms be implemented efficiently using available systems and methodologies? The answer requires an objective assessment of the difficulties inherent in the transition from problem conceptualization to physical reality. The implementation system, which provides this transition through successive interpretations of the problem solution, has a profound impact on performance that simply cannot be ignored.

The immediate effect of the implementation system is that it establishes a processing environment within which the problem solution is expressed. Two alternatives are currently available, sequential and applicative. They present conflicting views of the underlying architecture as von Neumann or not, but this is not their most important difference. The essence of the processing environment is that it establishes a computational paradigm which shapes the development and ultimate performance of any program executing within it.

The sequential processing environment views a problem solution in terms of the Turing model of computation, which isolates program control from data values in immutable and mutable stores, respectively. The procedural languages provide a natural expression of this

approach by treating program symbols as references to storage locations and limiting processing to the sequential or conditional execution of basic operations. In terms of performance improvement, the factors of greatest influence are early binding, which permits the propagation of values on a local and/or global level to minimize recomputation, and a heavy dependence on iteration, which allows the movement of instructions from more to less frequently executed portions of the control store. The major obstacles to optimization are the dangers posed by side effects and aliasing and the difficulty of recognizing potentially parallel operations.

The applicative processing environment, on the other hand, exemplifies the recursion theory approach to computability, viewing control and data elements uniformly as values. A symbolic language, which expresses processing in terms of functional composition and recursion, is the most natural descriptive tool for this model. Unlike their sequential counterparts, applicative programs permit no side effects or aliasing and possess an implicit concurrency which makes them admirable targets for parallelization. Their dependence on recursion and late binding, however, seriously hampers other attempts at optimization.

In terms of the implementation of AI algorithms, neither sequential nor applicative processing can be said to be unequivocally superior to the other. Each approach is inherently suited to a specific set of problems and inappropriate for others. This is reflected in the fact that virtually all existing algorithms were

developed within a particular environment and that transformations from one to the other are difficult and generally inefficient. The issue here is not the promotion of particular languages, implementation schemes, or computation systems. It is, rather, that the complex real-time systems required for AI processing in the BM/C³ setting combine some intrinsically sequential features with others that are intrinsically non-sequential. The use of a homogeneous implementation system is like choosing a broom instead of a shovel for a job that requires both: it can be done, but not efficiently.

Spanning dissimilar environments facilitates program improvement by allowing the assignment of subproblems on an individual basis to whatever system offers the best chance for automated optimization. At the same time, the strategy takes into account the fact that the number and speed of operations may have less effect on performance than design factors such as how pattern data is conceptualized or which heuristics guide allocation activities. A heterogeneous environment also maximizes this human optimization potential by allowing the designer/implementor to express each problem in the most natural way, without undue concern for the execution details of interacting solutions. It is to be hoped that an algorithm which in its entirety is too unwieldy for significant performance improvement can be reduced to a tractible level by this divide-and-conquer approach.

References

[Anklam 82]

P. Anklam, D. Cutler, R. Heinen Jr., and D. MacLaren. 1982. *Engineering a Compiler: VAX-11 Code Generation and Optimization*. Bedford: Digital Press.

[Brownsmith 84]

J. D. Brownsmith and L. H. Oliver. 1984. "Optimizing Loops in Programs Compiled with the IBM PL/I Optimizing Compiler." *ACM SIGPLAN Notices*, 19 (8):77-84.

[Bruynooghe 84]

M. Bruynooghe and L. M. Pereira. 1984. "Deduction Revision by Intelligent Backtracking". In J. L. Campbell, ed. *Implementations of PROLOG*. Chichester: Ellis Horwood. pp. 194-215.

[Carter 82]

L. R. Carter. 1982. *An Analysis of Pascal Programs*. Cited in W. M. Waite and G. Goos, eds. 1984. *Compiler Construction*. New York: Springer-Verlag. p. 354.

[Cocke 80]

J. Cocke and J. Markstein. 1980. "Measurement of Code Improvement Algorithms." *Information Processing*, 80: 221-228.

[Elshoff 76]

J. L. Elshoff. 1976. *An Analysis of Some Commercial PL/I Programs*. *IEEE Transactions on Software Engineering*, SE-2 (2): 133-120.

[Gabriel 85]

R. P. Gabriel. 1985. *Performance and Evaluation of Lisp Systems*. Cambridge: MIT Press.

[Goldbert 72]

P. C. Goldbert. 1972. "A Comparison of Certain Optimization Techniques." In R. Rustin, ed., *Design and Optimization of Compilers*. Englewood Cliffs: Prentice-Hall. pp. 31-50

[Knuth 71]

D. E. Knuth. 1971. "An Empirical Study of FORTRAN Programs." Software -- Practice and Experience, 1: 105-133.

[OTI 86]

Optimization Technology, Inc. 1986. "Support of the Militarized Computer Module Development". Unpublished technical report prepared for Control Data Corporation.

[Robinson 75]

S. K. Robinson and I. S. Torsun. 1975. An Empirical Analysis of FORTRAN Programs. Computer Journal, 19 (1): 56-62.

[Sarraga 84]

R. F. Sarraga 1984. Static Data Flow Analysis of PL/I Programs with the PROBE System. IEEE Transactions on Software Engineering, SE-10 (4): 451-459.

[Wulf 75]

W. Wulf, R. Johnsson, C. Weinstock, S. Hobbs, and C. Geschke. 1975. The Design of an Optimizing Compiler. New York: American Elsevier.

[Zelkowitz 76]

M. V. Zelkowitz. 1976. Automatic Program Analysis and Evaluation. Second International Conference on Software Engineering. pp. 158-163.

Appendix A

Optimization Techniques for Sequential Environments

The improvement techniques typically applied by optimizers in the sequential environment assume a bipartite immutable/mutable program organization as discussed in Chapter 3. A related assumption is that there be a clear distinction between symbols referencing program control units and those defining storage elements (program data versus problem data). Therefore, with few exceptions procedural languages require that user-defined symbols, or identifiers, be bound to the appropriate location in the control or data segment prior to execution. This property, which allows the association of symbols to locations through a static analysis of the program, is called early binding,¹ and is a major difference between sequential and non-sequential processing.

Prior to the application of optimizing transformations, control flow and data flow analysis are performed to establish the semantic framework (or "meaning") which must be preserved. The basic semantic element of a program, which we will call a logical unit, is a maximal collection of instructions in a textual sequence that are always

¹ "Early binding" is used here to indicate the pre-execution ability to associate symbols with offsets into a storage area. In the strictest sense, local variables are bound dynamically, since the storage area itself is not allocated until the unit's prologue is activated.

executed in order as a single entity.¹ This means that no transfer occurs to any instruction in a logical unit except the first and that once the first instruction is executed all others will be executed in sequential order prior to transfer out of the unit. A program is divided into logical units by identifying the statements which serve as entry (branch-in) and exit (branch-out) points.

In control flow analysis, the program is partitioned into logical units and a flow graph is constructed whose nodes are the units and whose arcs represent possible flow of control between units. The nodes are then grouped to form logical intervals, which represent sequential series of units dominated by a single entry node, but possibly having multiple exits. The importance of logical intervals is that they may be "reduced" to single nodes to form a new flow graph. This is then partitioned into new logical intervals which are subsequently reduced, and so forth, allowing the analysis of -- and hence the application of optimizing transformations to -- successively larger portions of the program.

Data flow analysis is concerned with the legitimate configurations of the data segment. Since D can be represented adequately by recording the changes brought about in each transition from D_n to D_{n+1} , the primary unit for data flow analysis is the state vector, which lists those data elements whose contents are altered by the

¹ Logical units are referred to elsewhere by a variety of names, including basic blocks, logical blocks, and control groups.

instruction corresponding to that state. Consolidated state vectors can be used to represent the data segment associated with each logical unit or interval of the program. Local data flow analysis concentrates on the state vectors of adjacent logical units, while global analysis correlates alterations to the data configurations which occur from one interval to another.

The sections which follow describe the performance improvement techniques that have been developed for use in sequential processing environments. The optimizing transformations typically applied to procedural programs are categorized according to the general type of analysis required for implementation. This is followed by a section addressing the question of how much performance can be expected to improve through the application of sequential optimizations.

A.1 Typical Optimization Techniques

The optimizations which are outlined in the following pages all presuppose at least a minimal level of control and data flow analysis. For convenience, they are grouped into four general categories¹: (1) expression simplification; (2) code rearrangement; (3) optimization of data storage areas; (4) target-specific optimizations. Examples are given of some of the most common techniques. Although they are

¹ Although there is no standard terminology for optimization techniques, an attempt has been made here to correlate terms from a variety of sources.

portrayed using a sort of pidgin-Pascal, it should be clear that most of the transformations are independent of any particular language structure or level of implementation. In general, individual techniques may be applied at the local and/or global level; the degree of analysis required for global optimizations, however, constrains their use in many settings.

The first class includes some of the most widely applied techniques (see Figure 15). Intuitively, expression simplification deals with improvements to the way in which numerical computations are specified. It encompasses a large subclass of transformations known as "constant folding" (also called compile-time computations or constant expression evaluation): the attempt to perform operations whose operands and/or results are known at compile time because they involve numerical constants. A second type of improvement, "common subexpression elimination" avoids the re-computation of values already calculated for some earlier operation. "Strength reduction" substitutes "weak" operations for "strong" ones to improve execution speed and/or make possible further optimizations; it includes attempts to reduce the processing needed to calculate array offsets. "Subexpression reordering" takes advantage of commutative and associative properties and algebraic identities to reduce temporary storage needs and to facilitate other transformations. Finally, "value propagation" eliminates or minimizes the need for storage transfers by replacing references to an identifier name by references to its value.

The concept of early binding is obviously crucial to these

Technique	Example
constant folding	<pre> CONST A = 5 B := (10*A)-5 ↓ B := 45 </pre>
common subexpression elimination	<pre> IF A*B > 10 THEN C := A*B ELSE C := -(A*B) ↓ temp := A*B IF temp > 10 THEN C := temp ELSE C := -temp </pre>
strength reduction	<pre> A := A*16 ↓ A := shift(A, left, 4) </pre>
subexpression reordering	<pre> A := A*D*B C := 2*(B*D) ↓ temp := D*B A := A*temp C := shift(temp, left, 1) </pre>
value propagation	<pre> A := 10*(B/2) C := B C := B*5 ↓ A, C := B*5 </pre>

Figure 15. Examples of Expression Simplification Techniques

techniques. An optimizer which applies expression simplifications must perform a detailed analysis of state vectors to determine when a value is altered, as well as incorporate mechanisms for associating algebraic properties with individual instructions. A specific transformation must normally be applied more than once and in alternation with other techniques to be truly effective.

The category of code rearrangement, as its name implies, encompasses transformations designed to improve the ordering of instructions in the program's code segment; some common examples are illustrated in Figure 16. The movement of invariant expressions out of loops, reordering of independent operations (statement flipping), elimination of induction variables, and hoisting of array offset calculations from inside loops all represent optimizations commonly referred to as "code motions". These techniques attempt to minimize the frequency with which a given operation is performed and are particularly important when a large number of array references are used, since offset calculations normally require costly multiplication operations. "Loop reorganizations" (linearization, fusion, and unrolling) reformulate loop structures by fully or partially expanding them to sequential form in order to minimize the number of tests and branches needed to control iteration. "Boolean minimization" performs a similar function by reordering comparisons in order to minimize testing.

Other types of code rearrangement are difficult to depict graphically. Code elimination techniques remove redundant instruc-

Technique	Example*
code motion	<pre>FOR I := 1 TO N DO BEGIN X := A[N] P[I] := ABS(B[I]) END</pre> <p style="text-align: center;">↓</p> <pre>X := A[N] FOR I := 1 TO N DO BEGIN A[I] := ABS(B[I])</pre>
	<pre>FOR I := 1 TO 10 DO BEGIN J := I+4 X[I] := B[1,J] Y[I] := B[1,J-2] END</pre> <p style="text-align: center;">↓</p> <pre>FOR offset := 0 TO 36 BY 4 DO BEGIN [X+offset] := [B+offset+16] [Y+offset] := [B+offset+8] END</pre>
loop reorganization	<pre>FOR I := 1 TO 10 DO FOR J := 1 TO 10 DO FOR K := 1 TO 10 DO READ(X[I,J,K])</pre> <p style="text-align: center;">↓</p> <pre>FOR offset := 0 TO 996 BY 4 DO READ([X+offset])</pre>
boolean minimization	<pre>IF A AND (B OR C) THEN X := 10 ELSE IF B OR C THEN X := 0 ELSE X := -10</pre> <p style="text-align: center;">↓</p> <pre>IF B THEN GOTO L1 IF C THEN GOTO L1 X := -10; GOTO L3 L1: IF A THEN GOTO L2 X := 0; GOTO L3 L2: X := 10 L3:</pre>

- * The use of "offset" is an attempt to indicate the calculation of array subscript offsets; a size of 4 bytes per element is assumed

Figure 16. Examples of Code Rearrangement Techniques

tions, such as the assignment of a value to a variable which is not referenced until after a further assignment, or unreachable (dead) code positioned after a branch-out point but before a corresponding branch-in. These transformations are often necessary after the application of other improvements. Because of the high run-time overhead associated with the prologue and epilogue code of procedure units, some optimizers also perform procedure integration (in-line substitution), which replaces each occurrence of an invocation by a copy of the instructions forming the body of the subprogram.

The third class of improvements, data storage optimizations, reorder the data segment to minimize program space requirements. The information encapsulated in the state vectors allows the identification and elimination of useless variables, such as unreferenced identifiers or those rendered extraneous through constant propagation or other optimizations. Live variable analysis reveals the effective span of individual identifiers so that variables with disjoint lifetimes may be overlaid in the same storage location. Time improvements can also be realized by reorganizing data elements to preserve boundary alignments which result in more efficient data access or to take advantage of reference adjacencies in order to minimize memory page faults. Finally, when performed in conjunction with expression simplification, storage analysis allows the replacement of run-time assignments of constant values by static (compile-time) initialization of the storage locations.

Target-specific optimizations are the most widely used tech-

niques since they are often directly incorporated in translation mechanisms. These include the use of improvement algorithms in such activities as register allocation, target instruction generation, and instruction scheduling. "Peephole optimization", applied during the last stages of target code generation, analyzes short sequences of code and attempts to reorder or eliminate instructions. For example, multiple instructions such as cascaded branches or redundant condition tests can be combined into a single operation having the same effect. The substitution of target-specific instructions which are shorter in format or execute faster is also of value at this level.

In summary, a wide range of techniques has been established for optimizing programs in the sequential processing environment. Unfortunately, some of the techniques are self-defeating -- if not actually contradictory -- when used in combination with others. An optimizer's effectiveness depends to a great extent on the successful interplay of a variety of techniques. Most existing versions limit their activities to a relatively small number of transformations sharing similar analysis needs and having significant impact on whatever types of input programs are deemed typical.

A.2 Potential for Optimization

Although optimizing compilers of varying degrees of sophistication have been available for twenty years, there are few empirical studies indicating to what degree they can improve performance. As

mentioned in Chapter 2, the first difficulty is establishing just what to measure -- i.e., what constitutes an average program, average run-time load, average input, etc. A second problem is how to isolate the effects of a particular improvement when the interaction of transformations is so critical to their success. The issue is further complicated by the general inadequacy of available methods for measuring run-time behavior and the unintelligibility of the results. In short, the literature is full of references to optimization techniques but there is a noticeable lack of correlation between theory and practice, and few statistically significant findings.

Knuth made the first attempt at compiling program statistics when he compared FORTRAN code written by Stanford students with that of Lockheed programmers, using both static and dynamic analysis techniques. This study [Knuth 71] remains one of the most extensive to date, but the results are of questionable use because of the heavy bias due to the syntax of early FORTRAN. [Elshoff 76] and [Sarraga 84] performed similar analyses of General Motors programs written in PL/I, while [Robinson 75] and [Zelkowitz 76] provide the best examples to date of academic programs (written in FORTRAN and PL/I, respectively). Since reasonably scaled analyses of other programming languages are not available, only those findings relevant to generally applicable optimization techniques will be cited here.

The mostly widely quoted statistic is Knuth's "90/10 rule", which stated that 90% of total time was spent executing just 10% of a program's statements. Input/output operations were found to consume

an inordinate share of processing, with 5% of the code accounting for more than 25% of measured time. In terms of non-I/O code, under 4% occupied 50% of execution. These figures imply that significant improvement might be realized if optimizing efforts can be concentrated in the proper areas.

Some of the statistics provided by static analysis are less encouraging. [Knuth 71] found that 68% of assignment statements were simple replacements which copied a value from one location to another; these results were confirmed by [Elshoff 76], who reported 77.6% and 40%, respectively. The same sources cite an additional 22%, 21%, and 29% as assignments involving the evaluation of no more than one operator. In terms of optimization potential, this indicates that even sophisticated expression simplification techniques may have negligible effects on performance. This view is confirmed by another researcher [Carter 82], who found that most blocks in Pascal programs included only two to four assignments and fewer than two common subexpressions. It seems realistic to estimate that while expression simplification and code rearrangement might save up to three-quarters of the time spent by numerical computation-intensive programs, the same techniques would probably show little effect on non-numeric programs.

The elimination of redundant assignments and useless variables seems more promising. [Elshoff 76] reported that of 384 identifiers in an average program, 10% were unreferenced. [Sarraga 84] performed a partial analysis of variable use which indicated that some 5% of

assignments were useless. At the same time, other figures compiled by Elshoff underscore the difficulty of performing the global data flow analysis needed for this type of optimization: he found that 13% of the gaps between successive references to a single identifier were more than 100 statements in length.

The implementors of optimizing compilers have on occasion published data indicating the degree of improvement measured by applying varying levels of optimization. Figure 17 illustrates the results cited by [Cocke 80] and [Brownsmith 84], compared with the improvements implemented manually by [Knuth 71]. The effects of the language-independent VAX-11 back-end optimizer designed by [Anklam 82] and currently used by the PL/I, C, and PEARL compilers, presented in Figure 18, were measured by inhibiting individual transformations on a series of benchmarks. [Wulf 75] attempted to quantify the effect on performance of each optimization performed by a Bliss-11 compiler (see Figure 19); the intention was to derive a formula expressing the cumulative result of varying combinations, but this did not prove to be practicable.

As the figures show, there is a significant range in performance from one optimizer to the next and from one benchmark to another. In some cases the effects of individual transformations almost escape measurement (the effects of loop invariant relocation and subexpression elimination on benchmark AB-5 in Figure 18, for example), while in others efficiency increases dramatically with the addition of a single technique (e.g., the effect of loop invariant relocation on

Technique	Measure	Level of Performance		
		Minimum	Average	Maximum
Local Optimizations ¹				
Knuth	time	40%	71%	91%
Cocke	time	32	50	72
	space	42	54	69
Brownsmith	time	15	63	99
Local plus Global Optimizations ²				
Knuth	time	11	28	91
Cocke	time	19	42	61
	space	38	55	66

1 Included local constant propagation, elimination of dead code, and local register allocation optimization

2 Added global constant propagation, strength and frequency reduction, and global register allocation optimization

Figure 17. Estimated Effects of Optimization

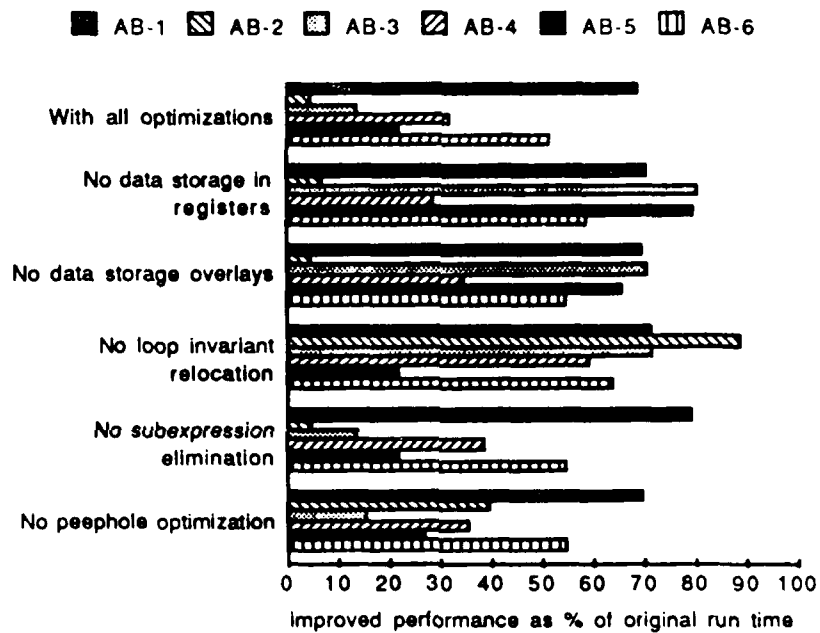


Figure 18. Anklaam's Benchmarks on the Effects of Optimization

Technique	Effect Factor
Constant folding	0.938
Common subexpression elimination	
- statement level	0.987
- local level	0.973
- global level	0.987
Algebraic laws	0.975
Code motion	0.985
Elimination of dead code	0.98
Register allocation	
- local	0.987
- global	0.975
Cross jumping	0.972
Peephole optimization	0.88

Figure 19. Wulf's Quantification of the Effects of Optimizing Techniques

AB-4 of the same figure). The extreme variability of these results illustrates the difficulty of realistically predicting the effects of isolated improvements.

No discussion of optimization would be complete without mention of parallelization. Recent years have seen an increasing interest in the development of translation algorithms for converting sequential programs to versions suitable for parallel processing. Available methods evolved from data flow analysis techniques and focus on array operations and looping structures as the primary candidates for parallelization. For example, the loop distribution algorithm for extracting parallel code assigns individual iterations of a loop to different processors. The pipelining algorithm, on the other hand, splits the loop into several component sub-loops, each of which is then assigned to a processor. In general, loop distribution is preferred when the loop body is small and the number of iterations large; pipelining is employed when the proportions are reversed. Unfortunately, a substantial amount of analysis is required to implement these techniques, nor are they uniformly applicable to all types of data elements and looping constructs. Furthermore, no conclusive empirical studies of the degree of improvement realized through parallelization have emerged to date.

AD-A192 848

IMPROVING THE PERFORMANCE OF AI ALGORITHMS(U) AUBURN
UNIV AL DEPT OF COMPUTER SCIENCE AND ENGINEERING
C H PANCAKE SEP 87 SCCE-PDP/85-48 RADC-TR-87-131

2/2

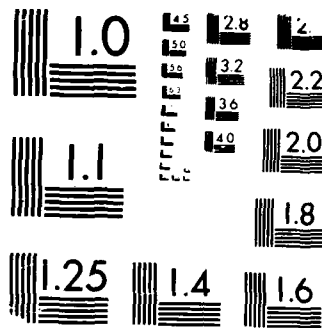
UNCLASSIFIED

F30602-81-C-0193

F/G 12/9

NL





MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A

Appendix B

Optimization Techniques for Applicative Environments

As pointed out in Chapter 3, optimizations based on normal control flow and data flow analysis are inappropriate in non-sequential environments and few optimizing transformations have as yet been developed specifically for applicative processing situations. Those that are available can be generally categorized as affecting either substitution or simplification activities.

The following sections discuss optimizations currently implemented in applicative environments. As in Appendix A, available techniques are described in general terms and then the results of studies examining the effectiveness of improvement activities are presented.

B.1 Typical Optimization Techniques

Substitution activities are optimized by improvements in heap storage management. A heap is difficult to implement efficiently, since it requires that a large, general-purpose storage area be made available for use on an unstructured, by-need basis. When a program element is defined, space is allocated to it from a free-space list and associated with the corresponding symbol by means of one or more levels of pointers. If the element is subsequently redefined, the

pointer or chain is altered to point to a new location. Note that there is theoretically no limit to the number of pointers which can reference the same object. The term garbage refers to a location which is no longer referenced by any pointer, and should therefore be placed on the free-space list. A dangling reference occurs when the object is returned prematurely to the free-space list, even though one or more pointers still refer to it. Traditional heap storage systems avoid dangling references by creating a unique object at each definition. This allows garbage to accrue rapidly; when the free-space list is exhausted, computation is suspended while a garbage collector searches the heap area, identifies garbage elements, and returns them to the list.

Garbage collection is clearly an attractive candidate for optimization. Established techniques include the use of hashed reference-count tables to keep track of active storage and/or the subdivision of storage into static, read-only, and heap areas, which somewhat reduces the area to be collected. Incremental ("on the fly") collection processes a small section of heap storage each time a specific operation is performed; this distributes the overhead more evenly over time but requires more space than other methods. "Compile-time garbage collection" attempts to replace some operations which create new definitions by altering the pointer links, but destructive changes of this type do not preserve equivalence with respect to multiple pointers referencing the same object.

A second type of optimization related to substitution is peculiar to LISP-based implementations. Most versions of LISP accommodate one or more special types of dynamic or "fluid" variable binding. With traditional binding (deep binding), each time the fluid variable is bound a tree of symbol tables must be searched to find the current value. The observation that the number of rebindings is small compared to the magnitude of the search led to a technique called shallow binding, whereby a current value cell is maintained for each name. The old value is stacked whenever a new instance is bound so that it can be restored easily when needed.

Additional techniques have been developed to improve simplification activities. Open coding involves the in-line expansion of common primitive functions and/or conditional structures (similar to procedure integration in the sequential environment). Calling sequence improvements are designed to expedite linkages between subprogram units. These make use of jump vectors, local branches, and linkage tables to eliminate calls to primitive linking routines.

A related method of diminishing simplification overhead is the removal of recursion. This is appropriate when the recursion is duplicated so that the same values are computed more than once; such techniques are similar to those for lazy evaluators (see below). A second use is in functions with "tail recursion", where the recursion is the last action of the current invocation. Since there is no need to establish a new application frame, the recursion is replaced by some form of open coding.

Other schemes improve the ways in which parameters are passed between functions, in an attempt to decrease the number of times a symbol is evaluated. Parameter rearrangements reorder arguments or place them in registers or on special parameter stacks. Call by name delays the evaluation of parameters until they are actually required, while lazy evaluation (call by need) maintains a table of values to avoid duplicate evaluations.

Arithmetic operations in the applicative environment are complicated by the need to convert numeric values to and from pointer representations. Common improvements include storing the values in registers, on special numeric stacks, or in tables. The complexity of numeric operations has also led to techniques similar to those used by sequential optimizers, such as constant folding, rearrangement, common subexpression elimination, and peephole optimization, but on a considerably smaller scale.

In summary, although some optimizations have been developed for the applicative environment, they are not as well understood as are the techniques discussed in the last chapter. Few compilers attempt to incorporate more than a handful of improvements and their interrelationships are only hazily defined. The most discouraging fact is that only a small percentage of existing implementations offer any significant degree of optimization.

B.2 Potential for Optimization

For the reasons outlined in previous sections, it comes as little surprise that no empirical studies of statistical significance have as yet appeared to establish the effectiveness of optimization in the applicative environment. Most recent research efforts have been directed instead to the development of architectures which process applicative programs directly rather than via software simulation. One recent study, however, compared the performance of a series of LISP implementations, including some employing improvement techniques [Gabriel 85].

Figure 20 summarizes the results of the Gabriel study on three systems using a Franz Lisp compiler. The only optimizations described are calling sequence improvements: the use of a *J(ump)S(u)B(routine)* instruction to perform direct jumps to functions included as part of the same compilation unit, and the incorporation of a transfer vector to replace the invocation primitive routine. It should be noted that all of the benchmarks tested were task-specific, and therefore are subject to the biases described in Section 2.2.

The results illustrate what appears to be a chronic problem with applicative optimizations. Although quite substantial improvements in run-time behavior are noted for some tests, performance is actually degraded in other cases. [Bruynooghe 84] reports similar results for the application to PROLOG programs of a technique called "intelligent backtracking". His tests were run on typical smallscale AI problems, with results that ranged from 0.3 to 219 (with an average of 112)

Benchmarks	Level of Performance *		
	Minimum	Average	Maximum
Recursion¹			
VAX 11/750	13%	32%	69%
VAX 11/780	13	29	60
Sun II	16	37	69
Knowledge Base²			
VAX 11/750	15	31	46
VAX 11/780	14	31	47
Sun II	18	37	56
Trees³			
VAX 11/750	15	53	80
VAX 11/780	14	52	81
Sun II	15	51	75
Searching⁴			
Sun II	99	99	99
IO⁵			
VAX 11/750	98	101	105
VAX 11/780	100	101	102
Sun II	99	109	126

* Garbage collection was excluded from the time calculation

1 Included the Tak, Stak, Ctak and Takt benchmarks

2 Boyer and Browae

3 Destructive, Traverse_Initialization, and Traverse

4 Puzzle

5 File_Print, File_Read, Terminal_Print

Figure 20. Gabriel's Benchmarks
on the Effects of Optimization

percent of the time required for the unimproved versions. This clearly violates the fundamental rule that an optimizing transformation be guaranteed at least not to adversely affect performance.

The shift from sequential to non-sequential architectures, on the other hand, seems encouraging. Any program performs significantly better once the software interpreting layers are eliminated from the applicative environment. The incorporation of parallelism will undoubtedly improve this situation even more, since the locality of effect and referential transparency properties of symbolic programs make them apt candidates for parallelization. Furthermore, the "generator" primitives of the functional languages (e.g., the MAP routines of LISP) are implicitly parallel constructs which can easily be adapted to concurrent processing situations. Finally, garbage collection activities have already been targeted for implementation on separate, dedicated processors, with the promise of substantial improvements in execution time.

Distribution List

Donald J. Gondek RADC/COES	8
RADC/DOVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	12
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
HQ USAF/SCTT WASHINGTON DC 20330	1
DIRECTOR DMAHTC ATTN: SDSIM 6500 Brookes Lane WASHINGTON DC 20315-0030	1
OASD (C3I), INFORMATION SYSTEMS ROOM 3E187 WASHINGTON DC 20301-3040	2
HQ AFSC/DLAE ANDREWS AFB DC 20334-5000	1

HQ AFSC/XRK
ANDREWS AFB MD 20334-500

1

HQ SAC/NRI (STINFO LIBRARY)
OFFUTT AFB NE 68113-5001

1

HQ SAC/SIPT
OFFUTT AFB NE 68113-5001

1

HQ ESC/DOOA
SAN ANTONIO TX 78243-5000

1

TAFIG/IIDD
ATTN: MR. ROBERTSON
LANGLEY AFB VA 23665-5000

1

HQ TAC/DOA (STINFO)
LANGLEY AFB VA 23665-5001

1

HQ TAC/DPCC
LANGLEY AFB VA 23665-5001

1

HQ TAC/DRCT
LANGLEY AFB VA 23665-5001

1

HQ AFOTEC (OAWD)
Attn: Capt. Novack)
KIRTLAND AFB NM 87117-7001

1

ASD/ENEGA
WRIGHT-PATTERSON AFB OH 45433

1

ASD/AXPM
WRIGHT-PATTERSON AFB OH 45433

1

ASD/AFALC/AXAE
WRIGHT-PATTERSON AFB OH 45433

1

AFIT/LDEE - TECHNICAL LIBRARY
BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

1

AFWAL/FIES/SURVIAC
WRIGHT-PATTERSON AFB OH 45433

1

AFAMRL/HE
WRIGHT-PATTERSON AFB OH 45433-6573

1

AFHRL/LRS-TDC
WRIGHT-PATTERSON AFB OH 45433-6503

1

Area A Technical Library
2750 ABW/SSLT
Bldg 266, Rm 207, Post 20311
Wright-Patterson AFB OH 454433

1

AFHRL/OTS
WILLIAMS AFB AZ 85240-6457

1

1843EIG/EIEXM
WHEELER AFB HI 96854

1

AUL/LSE 67-342
MAXWELL AFB AL 36112-5564

1

HQ SPACECOM/XPYX
ATTN: DR. WILLIAM R. MATOUSH
PETERSON AFB CO 80914-5001

1

HQ ATC/TTQI
RANDOLPH AFB TX 78148

1

HQ ATC/TTQE
RANDOLPH AFB TX 78148

1

CODE H396RL TECHNICAL LIBRARY
DEFENSE COMMUNICATIONS
ENGINEERING CENTER
1860 WIEHLE AVENUE
RESTON VA 22090

1

COMMAND CONTROL AND COMMUNICATIONS DIV
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION COMMAND
ATTN: CODE DIOA
QUANTICO VA 22134

1

AFLMC/LGY
ATTN: CH, SYS ENGR DIV
GUNTER AFS AL 36114

1

COMMANDER
BALLISTIC MISSILE DEFENSE SYSTEMS COMMAND
ATTN: DASD-H-MPL
PO BOX 1500
HUNTSVILLE AL 35807-3801

1

COMMANDING OFFICER
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46218

1

COMMANDING OFFICER
NAVAL TRAINING EQUIPMENT CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100

1

COMMANDER
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 9642
SAN DIEGO CA 92152-5000

1

US NAVAL WEAPONS CENTER, CODE 343
ATTN: TECHNICAL LIBRARY
CHINA LAKE CA 93555

1

SUPERINTENDENT (CODE 1424)
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5100

1

COMMANDING OFFICER
NAVAL RESEARCH LABORATORY
CODE 2627
WASHINGTON DC 20375

1

NAVELEXSYCOM
PDE-110-33
WASHINGTON DC 20363

1

REDSTONE SCIENTIFIC INFORMATION CENTER 2
US ARMY MISSILE COMMAND
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN: DRSMI-RPRD
REDSTONE ARSENAL AL 35898-5241

Advisory Group on Electron Devices 2
Hammond John/Technical Info Coordinator
201 Varick Street, Suite 1140
New York NY 10014

UNIVERSITY OF CALIFORNIA/LOS ALAMOS 1
NATIONAL LABORATORY
ATTN: DAN BACA/REPORT LIBRARIAN
P.O. BOX 1563, MS-P364
LOS ALAMOS NM 87545

RAND CORPORATION THE/LIBRARY 1
HELPER DORIS S/HEAD TECH SVCS
P.O. BOX 2138
SANTA MONICA CA 90406-2138

Commander 1
HQ, Fort Huachuca
TECH REF DIV
ATTN: BESSIE BRADFORD
Ft. Huachuca AZ 85613-6000

AEDC LIBRARY (TECH REPORTS FILE) 1
MS-100
ARNOLD AFS TN 37389-9998

JTFPMO 1
Attn: Technical Director
1500 Planning Research Drive
McLean VA 22102

AWS TECHNICAL LIBRARY 1
FL4414
SCOTT AFB IL 62225-5438

485 EIG/EIER (DMO) 2
GRIFFISS AFB NY 13441-6348

HQ ESD/XRX
HANSKOM AFB MA 01731

1

ESD/ICP
HANSKOM AFB MA 01731-5000

1

ESD/XRSE
HANSKOM AFB MA 01731-5000

1

ESD/TCS-10
ATTN: CAPTAIN J. MEYER
HANSKOM AFB MA 01731-5000

1

The Software Engineering Institute
Attn: Major Dan Burton, USAF
580 South Aiken Avenue
Pittsburgh PA 15232-1502

1

DIRECTOR
NSA/CSS
ATTN: T5112 /TDL (MARJORIE E. MILLER
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: W161
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R24
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R31
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R5
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: R8
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: S031
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: S21
FORT GEORGE G MEADE MD 20755-6000

1

DIRECTOR
NSA/CSS
ATTN: V307
FORT GEORGE G MEADE MD 20755-6000

1

DOD COMPUTER SECURITY CENTER
ATTN: C42, TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000

1

AUBURN UNIVERSITY
DEPARTMENT of COMPUTER SCIENCE & ENGINEERING
107 DUNSTAN HALL
AUBURN UNIVERSITY, ALABAMA 36849-3501

5

ESD-MITRE Software Center Library
X Ms J.A. Clapp
MITRE Corp D-70 MS A-359
Burlington Road
Bedford MA 01730

2

Software Engineering Institute Tech Library 2
Carnegie-Mellon University
Pittsburgh, PA 15232
ATTN: Korola Fuchs

Col J. Green 1
Dir, STARS JPO
Rm C-107
1211 South Fern Street
Arlington, VA 22202

SDIO/S-BM (Lt Col Audley) 1
The Pentagon
Washington DC 20301-7100

SDIO Library 1
IDA
1801 N. Beauregard St
Alexandria VA 22311

SAF/AQSD (Lt Col Harry Rosen) 1
The Pentagon
Washington DC 20330

AFSC/CV-D (Lt Col Ben Greenway) 1
Andrews AFB MD 20334-5000

HQ SD/XR (Col Peura) 1
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

SD/CN (Col Wilkenson) 1
PO BOX 92960
Worldway Postal Center
Los Angeles CA 90009-2960

ESD/MD (Col Paul) 1
Hanscom AFB MA 01731-5000

AFSTC/XLY (Lt Col Detucci)
Kirtland AFB NM 87117

1

USA SCC/DASD-H-SB (Larry Tubbs)
PO Box 1500
Huntsville AL 35807

1

ANSER Corp
Suite 800
Crystal Gateway 3
1215 Jefferson Davis Highway
Arlington VA 22202

1

IDA (Albert Perrella)
1801 N. Beauregard Street
Alexandria VA 22311

1

AFOTEC/XPP (Capt Wrobel)
Kirtland AFB NM 87117

1

AF Space Command/XPXIS
Peterson AFB CO 80914-5001

1

SDIO/S-BM (Capt Hart)
The Pentagon
Washington DC 20301-7100

1

SDIO/S-BP (Maj James Price)
The Pentagon
Washington DC 20301-7100

1

SDIO/S-BP (Maj Snow)
The Pentagon
Washington DC 20301-7100

1

SD/CHI (Col Hohman)
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

SD/CNIS (Lt Col Pennell)
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

SD/CNW/CWX/CNB
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

3

ESD/MDS (Lt Col Oldenberg)
Hanscom AFB MA 01731-5000

1

ESD/MDN (Lt Col Leib)
Hanscom AFB MA 01731-5000

1

DIR NSA (V42 Maj Morgan)
9800 Savage Road
Ft George Meade MD 20755-6000

1

END

DATE

FILMED

6-1988

DTic